Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Академии наук СССР

А.В. Климов, С.А. Романенко

МЕТАВЫЧИСЛИТЕЛЬ ДЛЯ ЯЗЫКА Р Е Ф А Л.

ОСНОВНЫЕ ПОНЯТИЯ И ПРИМЕРЫ

Препринт № 71 за 1987 г.

Москва

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М.В. Келдыша
Академии Наук СССР

Анд.В.Климов, С.А.Романенко

МЕТАВЫЧИСЛИТЕЛЬ ДЛЯ ЯЗЫКА РЕФАЛ.

ОСНОВНЫЕ ПОНЯТИЯ И ПРИМЕРЫ

Москва
1987

УДК 681.3.06

Описан проект метавычислителя - системы, выполняющей специализацию (частичные вычисления) всех вызовов функций в программе на языке Рефал, используя частичную определенность аргументов. Характерной чертой метавычислителя является наличие средств управления процессом специализации программы со стороны пользователя. Из известных методов специализации рефал-программ выделяется набор базовых средств, которые под управлением человека позволяют компилировать в Рефал программы на простых, но практически значимых языках с использованием в качестве описания семантики языка текст его интерпретатора на Рефале. Демонстрируется, что при наличии метавычислителя отпадает необходимость в макрогенераторе над Рефалом.

КЛЮЧЕВЫЕ СЛОВА И ФРАЗЫ: языки программирования, компиляция и оптимизация программ, специализация программ, частичные вычисления, генератор компиляторов, метавычислитель, Рефал.

СОДЕРЖАНИЕ

I. Введение

Прошло полтора десятилетия с тех пор, как японский математик Й.Футамура сообщил в [I] о подходе к генерации компиляторов, использующем в качестве описания семантики языка текст его интерпретатора на некотором алгоритмическом метаязыке. В течение 70-х годов этот результат несколько раз переоткрывался и к концу десятилетия стал достоянием программистской науки.

С тех пор появилась серия публикаций по методам, которые должны были превратить изящную математическую формулу в программный инструмент. Работы велись и ведутся во многих местах как в СССР, так и зарубежом. Но до сих пор эта тема остается достоянием "теоретического программирования", а не практического.

В чем же дело? Нам кажется, что помимо общих проблем превращения экспериментальных программ в программный продукт здесь сказалась установка авторов на автоматическое решение задачи, т.е. полностью машинное, алгоритмическое, а не на автоматизированное, т.е. выполняемое под управлением человека. Ситуация здесь осложнилась тем, что сразу не удалось найти компактную совокупность алгоритмических методов, которая покрывала бы основные нужды практики.

Но в действительности практика никогда и не требовала переложить всю работу с человека на ЭВМ. Даже наоборот: пользователи очень ценят системы, которые работают просто и понятно с использованием небольшого числа стройных алгоритмов, и содержат удобные "рычаги управления", выраженные какими-то языками, на которых можно сообщать системе "подсказки".

В этой работе мы выделим из совокупности известных методов преобразований и оптимизации программ на Рефале как можно более простую "базу", позволяющую алгоритмизировать процесс компиляции программ по тексту интерпретатора для простых, но содержащих основные типичные конструкции языков. При этом в тех случаях, когда "базового метода" не хватит для автоматического принятия решений, будет использо-

ваться дополнительная информация, подготовленная человеком. Это и есть основная наша цель: ввести и продемонстрировать систему понятий, в терминах которой человек сможет управлять процессом компиляции программы по описанию интерпретатора языка.

2. ПОСТАНОВКА ОБЩЕЙ ЗАДАЧИ

Результат Й.Футамуры основан на понятии *специализации* программ [2, 3]. Рассмотрим программу, описывающую функцию двух аргументов $F(X,Y)$. Если зафиксировать один аргумент, скажем X, положив его равным какой-то константе A, то получим функцию одного аргумента:

$$F1(Y) = F(A,Y) .$$

Такая функция $F1$ называется результатом *специализации* функции F по первому аргументу, равному A.

Программистов вряд ли устроит такое формальное построение одной программы из другой: с их точки зрения определение $F1$ "не достаточно эффективно", т.к. при каждом вызове $F1$ функция F будет заново выполнять все те операции, которые зависят только от $X=A$, а их может быть много.

С другой строны, аргумент-константа - это дополнительная информация о функции: используя ее, можно прооптимизировать программу в период компиляции. Простейшие образцы оптимизаций такого рода известны под названиями "вычисление константных подвыражений", "распостранение констант" и т.п. Многие компиляторы умеют выполнять хоть какую-нибудь работу из этой области.

Назовем *специализатором* программу SPEC, которая по описанию функции двух аргументов F на некотором метаязыке* M и значению A ее первого аргумента

---

* Мы называем язык, на котором программы подаются специализатору, *метаязыком* потому, что на нем же будет описываться интерпретирующая семантика языков.

строит описание функции одного аргумента FI такой, что для любого Y выполняется FI(Y) = F(A,Y) . Таким образом:

FI = SPEC(F,A) .

Конечно, при построении FI специализатор выполняет какие-то оптимизации, основанные на известности первого аргумента функции F .

Схема использования специализатора выражается следующим о с н о в н ы м с о о т н о ш е н и е м :

F(X,Y) = SPEC(F,X)(Y) ,

где перед левыми скобками подразумевается операция применения функции к аргументу.

Эта схема говорит о том, что вместо того, чтобы вычислять функцию двух аргументов, подав ей сразу два значения, можно взять только первый аргумент, провести по нему специализацию и, получив преобразованную функцию одного аргумента, применить ее ко второму значению. Ясно, что такой процесс будет выгодным, если первый аргумент "меняется редко", а второй - "часто". Именно так обстоит дело с использованием интерпретаторов.

Интерпретатор некоторого языка L - это функция двух аргументов:

L-INT(PROG-L,DATA) ,

где PROG-L - текст программы на интерпретируемом языке L , DATA - исходные данные для PROG-L .

Пусть L-INT запрограммирован на метаязыке M . Тогда можно проспециализировать интерпретатор L-INT по первому аргументу PROG-L :

L-INT(PROG-L,DATA) = SPEC(L-INT,PROG-L)(DATA) .

Видно, что результат специализации - это не что иное, как программа, эквивалентная PROG-L , переведенная на

метаязык M :

PROG-M = SPEC(L-INT,PROG-L) .

Сам специализатор - это тоже функция двух аргументов, запрограммированная на метаязыке M , поэтому можно применить основное соотношение несколько раз:

L-INT(PROG-L,DATA) = SPEC (L-INT, PROG-L)(DATA) =
= SPEC (SPEC, L-INT)(PROG-L)(DATA) =
= SPEC (SPEC, SPEC)(L-INT)(PROG-L)(DATA) =
= SPEC (SPEC, SPEC)(SPEC)(L-INT)(PROG-L)(DATA) .

В результате возникают:*

SPEC(L-INT,PROG-L) = PROG-M : DATA - REZULT

- скомпилированная с языка L
на метаязык M программа PROG-L ,

SPEC(SPEC, L-INT) = L-COM : PROG-L - PROG-M

- компилятор языка L в метаязык M ,

SPEC(SPEC, SPEC) = CO-GEN : L-INT - L-COM

- генератор компиляторов.

После применения того же соотношения в четвертый раз слева повторяется генератор компиляторов. Возможен вопрос: зачем применять CO-GEN к специализатору, снова получая CO-GEN ? Оказывается, что и это имеет смысл, когда нужно получить новую версия генератора компиляторов по новому специализатору методом "раскрутки":

---

* После двоеточия указано, что является аргументом и значением сформированной специализатором программы.

$$\begin{aligned} &\text{CO-GEN}\,(\text{SPEC}')(\text{SPEC}')(\text{SPEC}') = \\ = \;&\text{SPEC}\,(\text{SPEC},\,\text{SPEC})(\text{SPEC}')(\text{SPEC}')(\text{SPEC}') = \\ = \;&\text{SPEC}\,(\text{SPEC},\,\text{SPEC}')(\text{SPEC}')(\text{SPEC}') = \\ = \;&\text{SPEC}\,(\text{SPEC}',\,\text{SPEC}')(\text{SPEC}') = \\ = \;&\text{SPEC}'\,(\text{SPEC}',\,\text{SPEC}') = \\ = \;&\text{CO-GEN}' \,. \end{aligned}$$

В работе [1] Й.Футамура ввел понятие специализатора и выписал формулы для получения PROG-M и L-COM, не указав способа порождения CO-GEN. Возможность автоматического вычисления генератора компиляторов была обнаружена независимо разными авторами. По-видимому, первыми работами, в которых обсуждается этот вопрос, являются [4, 5 (стр.95), 6].

Для любителей "штурмовать крепости" самая увлекательная задача - это вычисление SPEC(SPEC,SPEC) для различных специализаторов. Совсем недавно появилось сообщение [8] о первом таком машинном эксперименте с содержательным результатом вычисления.

Но наша цель - выяснить, что можно внедрить в практику хоть каким-то решением первой задачи:

$$\text{PROG-M} = \text{SPEC}(\text{L-INT},\text{PROG-L}) \,.$$

Т.е. пока нас интересует только однократное применение специализатора - без самоприменения.

3. ПОНЯТИЕ МЕТАВЫЧИСЛИТЕЛЯ

Рассмотренный способ изображения идеи специализации программ удобен для компактной записи формул для PROG-M, L-COM и CO-GEN, но не слишком подходит для обсуждения практического применения простых специализаторов. Вернемся к первоначальному, "неэффективному" определению специализированной функции, которое получается добавлением к программе F предложения вида:

$$\text{F1}(Y) = \text{F}(A,Y) \,.$$

Эта фраза вместе с текстом функции F образует текст функции F1 .

Работа специализатора состоит в преобразовании программы F1 :

$$SPEC(F,A) = OPT(F1) ,$$

где OPT - это некоторый "оптимизатор" программ на метаязыке M .

Работу оптимизатора OPT естественно не связывать с какой-нибудь одной функцией типа F1 : если уж он умеет как-то специализировать функции, используя частичную определенность агументов, то пусть он делает это д л я в с е х в ы з о в о в функций в программе. В частности, он может оставить обращение к функции без изменения или - если удастся - вычислить его до конца, заменив константой.

Назовем оптимизатор, удовлетворяющий таким неформальным требованиям, м е т а в ы ч и с л и т е л е м .*

Этот термин следует понимать так: метавычислитель делает в каком-то смысле такую же работу, что и обычный "вычислитель", но при метавычислениях в качестве аргументов функций используются не настоящие значения, а символические обозначения классов значений: т.е. как-бы "то же самое", но на "метауровне".

Но как мы увидим, общие черты у конкретных вычислений и метавычислений имеются только в самых простых случаях. А чем

---

* В работах [6, 7] оптимизатор такого рода называется с у п е р к о м п и л я т о р о м . Подход его авторов отличается от нашего следующим образом: метавычислитель работает по несложным методам и должен управляться человеком там, где требуется принятие нетривиальных решений, а авторы суперкомпилятора стремятся как можно больше работы возложить на машину, поднимая "интеллектуальность" системы. Им удалось разработать ряд новых, очень интересных методов, часть которых заимствована нами для работ по метавычислителю.

дальше, тем больше для метавычислений требуется разработка своих методов.

4. ПЕРВЫЙ ПРИМЕР

Приступим к разбору примеров, на основе которых выясним, что, зачем и как может делать метавычислитель.

В качестве языка программирования используем язык Рефал [5].* Рефал создавался как метаязык [9]. Это повлияло на то, что в отличие от распространенных "практических" языков, у которых основная цель – дать программисту как можно больше готовых средств, система понятий Рефала в значительной степени минимизирована. Все недостающее считается специальными понятиями конкретных областей применения. Их надо определять через собственные понятия Рефала.

Основной способ определения на Рефале частных языковых конструкций – написание их интерпретаторов или – выражаясь в соответствии с рефальской традицией [10] – интерпретирующих функций.

В Рефале имеются целые числа и набор функций для выполнения арифметических операций, но нет понятия арифметического выражения. Определим его. Для наглядности примеров используем более простое и "менее эффективное" представление чисел, чем то, что имеется в реализациях Рефала.

В наших примерах целые числа будут изображаться последовательностями символов-литер – десятичных цифр, быть может, с символом '-' впереди.

Пусть имеется 3 функции арифметических операций сложения, вычитания и умножения с такими форматами:

---

* Предполагается, что читатель знаком с Рефалом в синтаксисе, принятом в его реализациях с 1972 г. (метакод-Б). Здесь используется небольшое изменение, появившееся только в последних трансляторах, например, на ЕС ЭВМ: функциональные скобки (знаки конкретизации) изображаются знаками "<" и ">", а не "К" и ".". Кроме того, разрешается опускать ограничители "/" у символов-меток, стоящих сразу после знака "<".

```
AREX
   R  EA '+' EB = <IADD (<AREX EA>) (<AREX EB>) >
   R  EA '-' EB = <ISUB (<AREX EA>) (<AREX EB>) >
   R  EA '*' EB = <IMUL (<AREX EA>) (<AREX EB>) >
      SF (EA) = < SF (<AREX EA>) >
      (EA)    = <AREX EA>
      EN      = EN
```

Пример использования:

```
FACT  ('0') = '1'
      (EN)  = <AREX (EN)'*'/FACT/((EN)'-1') >
```

Желаемая эффективная программа:

```
FACT  ('0') = '1'
      (EN)  = <IMUL (EN)                                    +
                    (<FACT (<ISUB (EN) ('1')>) >) >
```

Результат метавычислений:

```
FACT  ('0') = '1'
      (EN)  = <IMUL (<AREX EN>)                             +
                    (<FACT (<ISUB(<AREX EN>)('1')>) >) >
```

Метод: Частичные вычисления.

Рис.I. Интерпретация и компиляция простейшего языка AREX.

```
<IADD (EI) (EJ)> → EK
<ISUB (EI) (EJ)> → EK
<IMUL (EI) (EJ)> → EK
```

где EI , EJ , EK – целые числа.

Понятие а р и ф м е т и ч е с к о г о в ы р а ж е -

н и я определим функцией AREX , изображенной на рис.I. Пример обращения к ней:

<AREX ('12+1')'*2-1' > ⟶ '25'

Первые 3 предложения функции AREX - это определение знаков операций '+', '-' и '*' с учетом их ассоциативности и приоритета. Для простоты мы сделали '-' более приоритетной операцией, чем '+'. Указатель обратного отождествления R существенен для неассоциативного '-' : благодаря R отождествляется самый правый знак операции '-'. Например:*

```
    <AREX '1-2-3' >
2:  <ISUB (<AREX '1-2' >) (<AREX '3' >) >
2,6: <ISUB (<ISUB (<AREX '1' >) (<AREX '2' >) >) ('3') >
 ... '-4'
```

Указатель R поставен также у предложений, анализирующих знаки '+' и '*'. Это сделано для реализации общепринятого порядка вычисления слева направо и для единообразия со знаком '-'.

Четвертое предложение AREX - это расширение арифметического выражения операндами, обозначающими обращения к унарным функциям с таким форматом:

<F (EI)> ⟶ EJ

где F - имя функции, EI и EJ - целые числа. Это расширение позволяет. например, вместо:

<AREX '1+'<FACT(<AREX '2+3' >)> >

писать:

<AREX '1+'/FACT/('2+3') >

---

* Выписывая шаги рефал-машины или метавычислителя, слева будем указывать номер примененного предложения. Два номера означают, что сразу преобразованы 2 независимых функциональных терма.

Последнее выражение будет вычисляться следующим образом:

```
I:   <IADD (<AREX 'I' >) (<AREX /FACT/('2+3') >) >
6,4: <IADD ('I') (<FACT(<AREX '2+3' >) >) >
и т.д.
```

Предпоследнее предложение функции AREX - это "вход в скобки". А последнее - при правильном обращении к AREX - описывает случай, когда аргумент AREX - целое число без знака.

Итак, у нас есть интерпретатор некоторого "языка". Дадим ему название по имени интерпретирующей функции: я з ы к A R E X .

В качестве примера интерпретируемой программы возьмем классический пример - функцию факториала. Выберем для нее такой формат:

```
<FACT (EI)> == EJ
```

Возможностей первого варианта языка AREX хватит только на то, чтобы записать рекурсивную формулу для аргумента, не равного 0 . Но и это уже компактнее, чем вручную расписывать арифметическое выражение через IMUL и ISUB (см. рис.1).

Теперь посмотрим, что может сделать метавычислитель с вызовом функции AREX из второго предложения функции FACT . Он пытается выполнять шаги рефал-машины "в общем виде" так же, как это делает обычный рефал-интерпретатор. Если ему это не удается из-за того, что мешают переменные, он оставляет функциональный терм без изменения:

```
    <AREX (EN)'*'/FACT/((EN)'-I') >
3:  <IMUL (<AREX (EN)>) (<AREX /FACT/((EN)'-I')     >) >
5,4:<IMUL (<AREX EN>)   (<FACT( <AREX (EN)'-I'>    )>) >
2:  <IMUL (<AREX EN>)   (<FACT( <ISUB(<AREX(EN)>)          +
                                    (<AREX 'I'>)> )>) >
5,6:<IMUL (<AREX EN>)   (<FACT( <ISUB(<AREX EN >)          +
                                    ('I')      > )>) >
```

Последнее выражение метавычислитель подставляет в правую часть второго предложения FACT на место старого. В результате получаем программу, изображенную на рис.I.

Как видно, результат метавычислений еще не идеален. Начнем развитие примера после более точного описания устройства метавычислителя.

5. КАК РАБОТАЕТ ПРОСТОЙ МЕТАВЫЧИСЛИТЕЛЬ

Метавычислитель, как и обычная рефал-машина, имеет поле зрения и поле памяти. В п о л е п а м я т и загружается преобразуемая рефал-программа. В п о л е з р е н и я по-очереди помещаются все правые части предложений из поля памяти. В отличие от рефал-машины поле зрения метавычислителя содержит переменные. В простейшем варианте метавычислитель выбирает ведущий функциональный терм так же, как и традиционная рефал-машина: самый левый внутренний. Если его преобразовать не удается, этот функциональный терм оставляется без изменения и не учитывается при выборе следующего ведущего терма.

Алгоритм отождествления метавычислителя отличается от работы рефал-машины в одном месте: если в процессе просмотра поля зрения он встречает переменную или невычисленный функциональный терм, то выполнение шага заканчивается тем, что ведущий терм остается без изменения.

Когда метавычислитель не сможет выполнить шаг ни для одного функционального терма в поле зрения, полученное выражение подставляется в поле памяти на место исходного выражения. После этого в поле зрения загружается правая часть следующего предложения. Когда все предложения будут преобразованы, в поле памяти окажется результат метавычислений.

Для первичных* функций метавычислитель может иметь свой набор алгоритмов выполнения. Простейший практический способ

---

* П е р в и ч н ы м и мы называем функции, которые известны рефал-машине заранее и которые, тем самым, не определяются рефал-предложениями.

обработки метавычислителем вызовов первичных функций таков: когда аргумент полностью определен, вызывается обычная первичная функция рефал-машины; в противном случае функциональный терм оставляется без изменения. Но сверх этого, обращения к первичным функциям могут преобразовываться по каким-нибудь другим правилам, известным метавычислителю. Например, для использованных нами IADD , ISUB , IMUL имеет смысл выполнять такие замены:

```
<IADD ('0') (E1)>  =  E1
<IADD (E1) ('0')>  =  E1
<ISUB (E1) ('0')>  =  E1
<IMUL ('1') (E1)>  =  E1
<IMUL (E1) ('1')>  =  E1
```

и много чего еще, например: учитывать ассоциативность, дистрибутивность и т.п. Подчеркнем, что эти возможности связаны не с основными алгоритмами метавычислителя, а с его библиотекой первичных функций. Она может пополняться пользователем.

Приведенный способ работы метавычислителя корректен для "чистого" Рефала, т.е. когда не используются первичные функции с побочным эффектом. Дело в том, что при выполнении "меташага" невычисленные функциональные термы могут изменить свое взаимное расположение в правой части предложения. В "реальном" Рефале положение функциональных термов означает не только места, куда вставляются результаты, но и последовательность вычислений. Это значит, что практический метавычислитель должен за этим следить и при опасности изменения последовательности обращений к функциям с побочным эффектом либо прекращать преобразования, либо строить более сложную программу со вспомогательными функциями.

Другая тонкость связана с возможностью размножения невычисленных функциональных термов. В чисто функциональном языке это может привести лишь к повторению одних и тех же вычислений, а в реальном - к предыдущей проблеме. Но и для "чистого" Рефала метавычислитель не имеет права ухудшать эксплуатационные характеристики программы.

Описанный алгоритм метавычислений - это основа всех дальнейших методов специализации программ. Он является обобщением широко используемого в компиляторах метода вычисления константных подвыражений на случай, когда выражения хоть и содержат переменные, но они "не мешают". Будем называть этот уровень метавычислений *частичными вычислениями*. В следующих двух разделах посмотрим, как его можно использовать.

6. ОДИН МЕТОД ПРОГРАММИРОВАНИЯ ПРИ ИСПОЛЬЗОВАНИИ МЕТАВЫЧИСЛИТЕЛЯ

Посмотрим, что нас не устраивает в результате компиляции функции FACT на рис.1: в двух местах остался функциональный терм <AREX EN> вместо ожидаемого EN .

Как бы сделать метавычислитель "поумнее", чтобы он смог выдать то, что мы хотим? Ответ: а никак. В исходной программе для этого нет никакой информации. Функция FACT , определенная через AREX , имеет более широкую область определения, чем запрограммированная вручную; например, вычисление <FACT('1+2')> даст '6'.

Нам нужно как-то указать, что EN - это заведомо число. Самый надежный и общий способ сделать это - ввести соответствующее изобразительное средство в интерпретируемый язык. Добавим еще один вид операндов: ('"'EN) - где символ двойной кавычки в начале выражения в скобках обозначает, что дальше идет готовое значение, которое никак не нужно обрабатывать, а только выдать в качестве результата.

Функция AREX с добавленным предложением и измененная функция FACT показаны на рис.2. В качестве упражнения можете вручную выполнить "меташаги" и убедиться, что простой метавычислитель выдаст то, что нужно.

Использованный метод программирования интерпретатора можно охарактеризовать в общем виде так: *избегайте "умолчаний", т.е. анализа интерпретатором "тонкой" информации, извлекаемой сложными вычислениями; вместо этого вводите в интерпретируемый язык явные синтаксические указатели.*

```
AREX
   R  EA '+' EB = <IADD (<AREX EA>) (<AREX EB>) >
   R  EA '-' EB = <ISUB (<AREX EA>) (<AREX EB>) >
   R  EA '*' EB = <IMUL (<AREX EA>) (<AREX EB>) >
      SF (EA) = < SF (<AREX EA>) >

      ('"'EA) = EA

      (EA)    = <AREX EA>
      EN      = EN
```

Пример использования:

```
FACT  ('0') = '1'
      (EN)  = <AREX ('"'EN)'*'/FACT/(('"'EN)'-1') >
```

Результат метавычислений:

```
FACT  ('0') = '1'
      (EN)  = <IMUL (EN)                                    +
                    (<FACT (<ISUB (EN) ('1')>) >) >
```

Метод: Частичные вычисления.

Рис.2. Интерпретация и компиляция языка AREX
с возможностью явно помечать числовые операнды.

7. ИНТЕРПРЕТАЦИЯ И КОМПИЛЯЦИЯ ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ

Уж очень громоздко пришлось изображать на рис.2 то, что математики привыкли обозначать одной буквой: ('"'EN) вместо, например, 'N' . Но ведь все в наших руках. Давайте интерпретировать!

Введем в язык AREX аппарат обозначений. Используем

```
AREX  EV ':=' EA ';' EB =
                <AREX <SUBST                                    +
                     ((EV '->' ('"' <AREX EA>))) EB             +
                >>

   R  EA '+' EB = <IADD (<AREX EA>) (<AREX EB>) >
   R  EA '-' EB = <ISUB (<AREX EA>) (<AREX EB>) >
   R  EA '*' EB = <IMUL (<AREX EA>) (<AREX EB>) >
      SF (EA)   = < SF (<AREX EA>) >
      ('"'EA)   = EA
      (EA)      = <AREX EA>
      EN        = EN
```

Пример 1:

```
FACT  ('0') = '1'
      (EN)  = <AREX 'N:='('"'EN)';'                             +
                    'N*'/FACT/('N-1') >
```

Результат метавычисления: оптимальный – см. рис.2.

Пример 2:

```
SUMSQ (EX) (EY) = <AREX 'X:='('"'EX)';'                         +
                        'Y:='('"'EY)';'                         +
                        'X*X+Y*Y' >
```

Результат метавычислений:

```
SUMSQ (EX) (EY) = <IADD (<IMUL (EX) (EX)>)                      +
                        (<IMUL (EY) (EY)>) >
```

Метод: Частичные вычисления.

Рис.8. Интерпретация и компиляция локальных переменных.

```
SUBST WS =

   R  WS S(L)U E(LD)V EA = <SUBST-IDENT WS SU EV>          +
                           <SUBST WS EA>

      WS SX EA           = SX                              +
                           <SUBST WS EA>

      WS ('"'EX) EA      = ('"'EX)                         +
                           <SUBST WS EA>

      WS (EA) EB         = (<SUBST WS EA>)                 +
                            <SUBST WS EB>

SUBST-IDENT
      (EI (EV'->'EX) E2) EV = EX
      (EI)               EV = EV
```

Формат функции SUBST :

<SUBST (ES) EA> → EB

где EB - измененное выражение EA ,
ES - подстановка - последовательность термов вида:
(идентификатор '->' выражение)

Во втором предложении используются спецификаторы Рефала-2:
S(L)U - это символ-буква,
E(LD)V - это выражение из букв и цифр.

Переменная E(LD)V принимает максимальное по длине значение благодаря указателю обратного отождествления R .

Рис.4. Функция подстановки вместо идентификаторов произвольных выражений.

традиционный синтаксис оператора присваивания:

```
'N:='('*'EN)';'
'N*'/FACT/('N-1')
```

Точку с запятой проинтерпретируем как операцию подстановки, преобразующую следующую за ';' часть текста программы перед ее вычислением. Слева от точки с запятой указывается описание подстановки, состоящее из идентификатора и арифметического выражения, разделенных знаком ':=' (эту часть условно назовем "оператором присваивания"). Такая конструкция выполняется следующим образом: сначала вычисляется значение правой части "оператора присваивания", затем полученное значение (обозначим его EX) подставляется в текст программы после ';' вместо указанного идентификатора в виде ('*'EX) - здесь двойной кавычкой отмечено, что EX пересчитывать не нужно. После этого продолжается интерпретация модифицированной части программы.

Соответствующая функция AREX изображена на рис.3. Новое предложение с левой частью EV':='EA';'EB добавлено первым, т.к. разделители ':=' и ';' имеют более низкий приоритет, чем '+', '-' и '*'. Используется функция подстановки SUBST , описанная и прокомментированная на рис.4. Она умеет выполнять одновременную подстановку значений нескольких идентификаторов, но пока мы используем ее только для одного. Функция SUBST учитывает синтаксис языка AREX только в одном месте: четвертое предложение функции SUBST выдает без какого-либо анализа те выражения, которые отмечены двойной кавычкой и заключены в скобки. Этого достаточно, чтобы обращения к функции SUBST вычислились полностью в период метавычислений (при условии, что в обращениях к функции AREX переменные и функциональные термы встречаются только внутри скобок, отмеченных знаком '*').

Пример 2 на рис.3 демонстрирует многократное использование "оператора присваивания". Функция SUMSQ ("сумма квадратов") содержательно имеет два аргумента, которые выделяются левой частью рефал-предложения, имеющей вид (EX)(EY) . Два присваивания позволяют ввести однобуквенные обозначения для обоих аргументов.

В этих примерах использованы простейшие правые части присваиваний. Но интерпретатор AREX позволяет ставить здесь произвольные арифметические выражения, содержащие обращения к другим функциям и даже вложенные присваивания (заключенные в скобки).

Этим мы заканчиваем обсуждение мощности простого метавычислителя, выполняющего в общем виде те шаги, в которых используется информация, представленная только константной частью аргументов. Этого оказалось достаточно, чтобы в интерпретируемом языке ликвидировать один из заметных недостатков такого "рафинированного" функционального языка, каким является Рефал, а именно: невозможность вводить локальные обозначения для промежуточных значений.

8. ИНТЕРПРЕТАЦИЯ И КОМПИЛЯЦИЯ РАЗБОРА СОСТАВНОГО ЗНАЧЕНИЯ

Предыдущие примеры запрограммированы как-бы "на смеси" собственно Рефала и интерпретируемого языка AREX . Пока возможностей одного языка AREX не достаточно, чтобы полностью запрограммировать такую функцию как FACT . Пойдем по такому пути развития языка AREX : "переложим" в него часть изобразительных средств из Рефала. Это позволит уменьшить "рефальскую часть" примеров. Одновременно надо будет увеличивать мощность метавычислителя так, чтобы он мог по-прежднему компилировать язык AREX обратно в Рефал.

Один шаг уже сделан: в язык AREX введены переменные. В отличие от Рефала их можно об'являть даже локально в подвыражениях.

Теперь расширим левые части "операторов присваивания" образцами в духе левых частей рефал-предложений. Аппарат отождествления используется в Рефале для двух целей: во-первых, для проверки условий и выбора пути дальнейших вычислений, во-вторых, для разбора аргумента на части и присваивания значений переменным. Позаимствуем для языка AREX только безусловный разбор на части по скобочной структуре.

Левая часть присваивания будет о б р а з ц о м с та-

ким синтаксисом:

```
образец = пусто
        | идентификатор
        | ( образец ) образец
```

Естественное применение образцов – выделение аргументов функции. Например:

```
SUMSQ EI = <AREX ('X')('Y')':='('"'EI)';'                    +
                 'X*X+Y*Y' >
```

Новая интерпретирующая функция AREX показана на рис.5. Она изменена в одном месте: на функцию SUBST подается подстановка, описывающая значения не одной – как на рис.3. – а нескольких переменных. Такая подстановка формируется функцией MATCH – "отождествление образца с выражением". Ее описание имеет 3 предложения, отражающие 3 случая в нашем определении синтаксиса образца.

Что же теперь сможет сделать метавычислитель? Рассмотрим обращение к AREX из SUMSQ :

```
<AREX ('X')('Y')':='('"'EI)';X*X+Y*Y' >
```

Выполняя частичные вычисления, он остановится в таком состоянии:

```
<AREX <SUBST (<MATCH (('X')('Y')) (EI) >) 'X*X+Y*Y' >>
```

Затруднения возникли с обращением к функции MATCH :

```
<MATCH (('X')('Y')) (EI) >
```

Проводя отождествление по стандартному алгоритму, метавычислитель определил, что первое и второе предложения не применимы, в третьем – отождествился первый терм (с такими значениями переменных: EV='X' , EW=('Y')), а дальше при попытке сопоставить (EA)EB с EI он остановился.

Но ведь третье предложение – последнее, и нам-то ясно, что, чтобы не произошло авоста, EI обязано иметь вид,

```
AREX  EV ':=' EA ';' EB =
              <AREX <SUBST                                    +
                   (<MATCH (EV) (EA)>) EB                     +
              >>

   R  EA '+' EB = <IADD (<AREX EA>) (<AREX EB>) >
   R  EA '-' EB = <ISUB (<AREX EA>) (<AREX EB>) >
   R  EA '*' EB = <IMUL (<AREX EA>) (<AREX EB>) >
      SF (EA)   = < SF (<AREX EA>) >
      ('''EA)   = EA
      (EA)      = <AREX EA>
      EN        = EN

MATCH () () =
      (SU EV)   (EA)       = (SU EV '->' ('''EA))
      ((EV) EW) ((EA) EB)  = <MATCH (EV) (EA)>                 +
                             <MATCH (EW) (EB)>
```

Пример 1:

```
FACT  ('0') = '1'
      EI    = <AREX ('N')':=('''EI)';'                         +
                    'N*'/FACT/('N-1') >
```

Пример 2:

```
SUMSQ EI = <AREX ('X')('Y')':=('''EI)';'                       +
                 'X*X+Y*Y' >
```

Результаты метавычислений: оптимальные - см. рис. 2 и 3.

Метод: Частичные вычисления и
прогонка единственных сужений.

Рис.5. Интерпретация и компиляция разбора составного значения на части по скобочной структуре.

который можно изобразить как (E2)E3 . Выполнив подстановку E1→(E2)E3 , можно продолжать работу дальше. После очередных 2 шагов вычисления прекратятся в таком состоянии:

```
('X->'('*'E2)) <MATCH (('Y')) (E3) >
```

Рассуждая точно так же, заменяем E3 на (E4) E5 . Еще через 2 шага получим:

```
('X->'('*'E2)) ('Y->'('*'E4)) <MATCH () (E5) >
```

Теперь останов произошел при отождествлении первого предложения. Но нам видно, что второе и третье предложения не применимы. Если это "увидит" и метавычислитель, то он сможет положить E5 равным пустому выражению, как это диктует первое предложение. После этого метавычисление всей правой части функции SUMSQ дойдет до того же состояния, что и в предыдущих примерах:

```
<IADD (<IMUL (E2) (E2)>) (<IMUL (E4) (E4)>) >
```

Этот результат был получен после такой серии подстановок:

```
E1  →  (E2) E3
E3  →  (E4) E5
E5  →  пусто
```

или в совокупности:

```
E1  →  (E2) (E4)
```

При замене правой части на итог метавычислений эти подстановки нужно применить и к левой части рефал-предложения. В результате получим то, что и хотели:

```
SUMSQ (E2) (E4) = <IADD (<IMUL (E2) (E2)>)                    +
                        (<IMUL (E4) (E4)>) >
```

На рис.5 показан результат аналогичных метавычислений

для новой версии функции FACT .

Остался один вопрос: какой формальный метод позволит метавычислителю воспроизводить наши рассуждения?

9. ПРОГОНКА

Нужный нам формальный метод преобразования рефал-программ впервые был описан в [11], а в [12] был назван п р о г о н к о й . Идея его состоит в следующем.

Обращение к функции из правой части рефал-предложения рассматривается как множество конкретных вызовов, получающихся подстановками вместо переменных всевозможных их значений. Это множество при выполнении шага рефал-машины распадается на подмножества, элементы каждого из которых "проходят" через одно и то же предложение.

Нас интересуют подмножества, имеющие конструктивные представления. Самый простой вид конструктивных подмножеств - это те, которые образуются подстановками вместо переменных некоторых выражений (быть может, снова содержащих переменные). Такие подмножества (и формирующие их подстановки) в [11, 12] называются с у ж е н и я м и .

Преобразование рефал-программы, называемое прогонкой, состоит в следующем. По выражению с переменными, стоящему в обращении к функции, и набору левых частей описания функции строится конечный набор сужений таких, что для каждого можно выполнить шаг рефал-машины в общем виде по одному из предложений. Предложение, содержащее обращение к функции, заменяется на несколько "суженных". В каждом из них обращение к функции заменяется на результат выполнения шага. (Это преобразование можно выполнить лишь при некоторых ограничениях на вид левых частей. Самый строгий вариант: запрещаются открытые E-переменные и повторные E- и W-переменные.)

Основа прогонки - алгоритм о б о б щ е н н о г о о т о ж д е с т в л е н и я произвольного выражения, содержащего переменные, с левой частью рефал-предложения. При отмеченных выше ограничениях он вычисляет пересечение этих выражений, представленное в виде об'единения конечного числа

сужений. В частности, этот алгоритм можно использовать для получения ответов на такие вопросы:

(1) пересекаются ли выражения;

(2) отождествляется ли выражение целиком с левой частью предложения и каковы значения переменных из левой части, полученные в результате отождествления;

(3) представимо ли пересечение выражений в виде одного сужения и какого вида.

Умения отвечать на эти вопросы достаточно, чтобы алгоритмизировать использованные в примерах методы метавычислений.

Метод частичных вычислений можно даже обобщить и сформулировать так: если выражение, стоящее в обращении к функции, не пересекается с левыми частями нескольких первых предложений, а со следующим отождествляется целиком, то выполняем шаг прогонки.

Компиляция функций типа MATCH обеспечивается такой стратегией применения правила прогонки: если выражение, стоящее в обращении к функции не пересекается со всеми левыми частями описания функции кроме одного, а пересечение с оставшейся левой частью описывается единственным сужением, то выполняется шаг прогонки. Это правило назовем прогонкой единственных сужений.

Рассмотренные два правила применения прогонки хороши тем, что они не увеличивают числа предложений. А гарантируют ли они завершение процесса метавычислений за конечное число шагов? Такой вопрос закономерен потому, что прогонку в общем случае можно применять до бесконечности, неограниченно увеличивая текст программы.

Заведомых гарантий нет и в этих частных случаях прогонки, т.к. метавычисления при полностью определенном аргументе функции превращаются в обычные вычисления, которые могут и "зацикливаться".

Тем не менее, частичные вычисления и прогонка единственных сужений всегда завершаются для "правильных" программ – правильных в следующем смысле: для правой части каждого предложения рефал-программы существует набор значений переменных, при котором ее вычисление доходит до нормального

конца (без авоста).

Это достаточное условие завершения метавычислений не так уж трудно проверять. Сравним его с общепринятым правилом минимального тестирования: "при выполнении набора тестов управление должно побывать в каждой точке программы". Для Рефала точки программы - это предложения. Поэтому правило тестирования для Рефала переформулируется так: "каждое предложение должно примениться хотя бы один раз". Это как раз то, что нужно, чтобы частичные вычисления не могли продолжаться без конца. А если к этому правилу добавить: "и все тесты должны завершиться нормальным остановом", то получим достаточное условие останова метавычислителя, использующего правило прогонки единственных сужений.

Отметим, что такое свойство метавычислений дает дополнительное обоснование для включения в реализации Рефала средств профилирования* программы.

10. ИНТЕРПРЕТАЦИЯ И КОМПИЛЯЦИЯ УСЛОВНОГО ВЫРАЖЕНИЯ

Методы метавычислений, использованные в предыдущих примерах, не требовали никакой дополнительной информации от человека. Но прогонка, как было отмечено, в общем случае "неустойчива": может применяться до бесконечности. Тем не менее, обращения к некоторым функциям имеет смысл прогонять даже с ростом числа предложений. Информацию о том, какие функции нужно прогонять, может сообщить метавычислителю человек.

Расширим язык AREX условными выражениями со следующим синтаксисом:

/IF/(ар-выр)/THEN/(ар-выр)/ELSE/(ар-выр)

где ар-выр - произвольное выражение языка AREX .

Для реализации условных выражений понадобится еще один

---

* Профилированием программы называется сбор статистики о частоте выполнения операторов.

```
AREX  EV ':=' EA ';' EB =                                          +
              <AREX <SUBST (<MATCH (EV) (EA)>) EB >>

   R  EA '=' EB = <EQU  (<AREX EA>) (<AREX EB>) >

   R  EA '+' EB = <IADD (<AREX EA>) (<AREX EB>) >
   R  EA '-' EB = <ISUB (<AREX EA>) (<AREX EB>) >
   R  EA '*' EB = <IMUL (<AREX EA>) (<AREX EB>) >

      /IF/(EP)/THEN/(EA)/ELSE/(EB) =                               +
              <AREX <IF (<AREX EP>) (EA) (EB) >>

      SF (EA) = < SF (<AREX EA>) >
      ('''EA) = EA
      (EA)    = <AREX EA>
      EN      = EN

IF    ('T') (EA) (EB) = EA
      ('F') (EA) (EB) = EB

EQU   (EX) (EX) = 'T'
      (EX) (EY) = 'F'
```

Пример использования:

```
FACT  EI = <AREX ('N')':='('''EI')';'                              +
                 /IF/('N=0')/THEN/('1')                            +
                 /ELSE/('N*'/FACT/('N-1')) >
```

Результат метавычислений: оптимальный – см. рис.2.

Метод: Частичные вычисления, прогонка единственных сужений и всех вызовов указанных программистом функций (здесь прогоняется EQU).

Рис.6. Интерпретация и компиляция условного выражения.

вид данных: б у л е в с к и е з н а ч е н и я . Выберем для них символы-литеры 'T' и 'F'. Добавим еще одну операцию '=' - сравнение.

Новая версия интерпретирующей функции AREX изображена на рис.6.

Рассмотрим процесс метавычислений для нового определения функции FACT , использующего условное выражение (рис.6). Проводя преобразование правой части функции FACT по "надежным" правилам частичных вычислений и прогонки единственных сужений метавычислитель остановится в таком состоянии:

```
<AREX <IF (<EQU (EN) ('0')>)
          ('1')
          (('*'EN)'*'/FACT/(('*'EN)'-1') >>
```

Если теперь попытаться прогнать обращение к функции EQU , то возникнут два сужения:

EN → '0' - по первому предложению,
пустая подстановка - по второму предложению.

Что получится, если мы попросим метавычислитель всегда прогонять обращения к функции EQU ? Тогда одно предложение функции FACT будет заменено на два таких:

```
FACT  ('0') = <AREX <IF ('T') ('1')                              +
                        (('*'EN)'*'/FACT/(('*'EN)'-1') >>

      (EN)  = <AREX <IF ('F') ('1')                              +
                        (('*'EN)'*'/FACT/(('*'EN)'-1') >>
```

После этого в каждой правой части можно продолжать частичные вычисления. В конце концов получим то, что хотели:

```
FACT  ('0') = '1'
      (EN)  = <IMUL (EN) (<FACT( <ISUB(EN)('1')> )>) >
```

II. ЗАКЛЮЧЕНИЕ

Были рассмотрены методы специализации обращений к функциям в рефал-программе, на основе которых строится метавычислитель, имеющий практическое значение.

Основой методов является прогонка [11, 12]. Прогонка - это правило преобразования программ, а не алгоритм оптимизации. В общем случае прогонку можно выполнять до бесконечности, неограниченно увеличивая размер программы. Тем самым, для построения алгоритмов оптимизации необходимо сформулировать стратегии, определяющие, когда применять прогонку.

Для метавычислителя предложено использовать две простые стратегии, которые могут выполняться автоматически. Для более сложных преобразований программы метавычислитель требует задания дополнительной информации человеком.

Этими автоматическими стратегиями являются:

(1) "частичные вычисления" - выполнение шагов рефал-машины, пока не потребуется информация о значениях переменных;

(2) "прогонка единственных сужений" - выполнение шага прогонки, когда в результате не порождается новое разветвление в программе.

Простейшим способом ручного управления процессом метавычислений является указание имен функций, которые нужно прогонять всегда.

На примерах было продемонстрировано, что таких методов достаточно для компиляции по описанию интерпретатора простых языков, содержащих:

- выражения по типу арифметических;
- локальные переменные и (ограниченный) оператор присваивания;
- условные конструкции.

Общей чертой этих элементов языков является то, что их компиляция не требуют порождения новых рекурсивных функций. Для решения последней задачи (возникающей, например, при

компиляции цикла) в метавычислитель будет введен еще один уровень управления от человека. Этот уровень уже не будет сводиться к прогонке, т.к. это преобразование рефал-программ не увеличивает числа функций.

Рассмотренные примеры показывают, что возможность компиляции интерпретируемых языков с помощью метавычислителя позволяет при программировании на Рефале вводить в него недостающие языковые конструкции, т.е. делать то, что в существующей практике программирования обеспечивается макрогенераторами. При этом имеется принципиальная разница в способе описания новых конструкций: при использовании макрогенератора человек пишет алгоритмы компиляции на некотором "макроязыке", обычно качественно отличающемся от расширяемого ("базового") языка: при использовании же метавычислителя нужно запрограммировать интерпретатор на том же самом языке (в нашем случае - на Рефале). У последнего способа имеются явные преимущества:

- во-первых, как известно программистам, интерпретатор запрограммировать гораздо проще, чем компилятор;
- во-вторых, отладку программы, содержащей обращения к интерпретирующим функциям, можно провести до преобразования программы метавычислителем, а макрогенерацию нужно производить перед запуском программы на счет после каждого ее изменения;
- в-третьих, метавычислитель всегда выдает заведомо правильную программу.

Следует отметить и один недостаток метавычислителя перед макрогенератором: с помощью метавычислителя можно сгенерировать не любую программу, которую можно сформировать макрогенератором; класс генерируемых программ зависит от "мощности" метавычислителя, в то время как при использовании макрогенератора все зависит от искусства пользователя.

В целом метавычислитель является промежуточной ступенью на пути по автоматизации построения компиляторов по интерпретирующей семантике для богатых, практических языков.

БЛАГОДАРНОСТЬ

Авторам приятно поблагодарить участников семинара Рабочей группы по Рефалу за плодотворное обсуждение материалов этой работы: С.М.Абрамова, Р.Ф.Гурина, А.С.Зиброва, Арк.В.Климова, Н.В.Кондратьева, А.Ю.Романенко, В.Я.Рудного и, особенно, В.Л.Кистлерова, прочитавшего несколько вариантов рукописи в процессе ее подготовки и высказавшего много ценных замечаний.

ЛИТЕРАТУРА

1. Y.FUTTAMURA. PARTIAL EVALUATION OF COMPUTATION PROCESS - AN APPROACH TO A COMPILER COMPILER. - SYSTEMS COMPUTERS CONTROLS, VOL.2, No.5, 1971, pp.45–50.

2. J.DIXON. THE SPECIALIZER, A METHOD OF AUTOMATICALLY WRITING COMPUTER PROGRAMS. - DIVISION OF COMPUTER RESEARCH AND TECHNOLOGY, NATIONAL INST. OF HEALTH, BETHENDA, MARYLAND, 1971.

3. CH.CHANG, R.LEE. SYMBOLIC LOGIC AND MECHANICAL THEOREM PROVING. - ACADEMIC PRESS, 1973. (Имеется русский перевод: Ч.Чень, Р.Ли. Математическая логика и автоматическое доказательство теорем: Пер. с англ./Под ред. С.Ю.Маслова. - М.: Наука, 1983. - 360 с.)

4. L.BECKMAN, A.HARALDSON, O.OSKARSSON, E.SANDEWAL. A PARTIAL EVALUATOR, AND ITS USE AS PROGRAMMING TOOL. - ARTIFICIAL INTELLIGENCE, VOL.7, No.4, 1976, pp.319–357.

5. Базисный Рефал и его реализация на вычислительных машинах. - М.: ЦНИПИАСС, 1977, вып. V-40, 258 с.

6. V.F.TURCHIN. SUPERCOMPILER SYSTEM BASED ON THE LANGUAGE REFAL. - SIGPLAN NOTICES, VOL.14, No.2, 1979, pp.46–54.

7. V.F.TURCHIN, R.N.NIRENBERG, D.V.TURCHIN. EXPERIMENTS WITH THE SUPERCOMPILER. - CONFERENCE RECORD OF THE ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, 1982, pp.47-55.

8. N.D.JONES, P.SESTOFT, H.SONDERGAARD. AN EXPERIMENT IN PARTIAL EVALUATION: THE GENERATION OF A COMPILER COMPILER. - SIGPLAN NOTICES, VOL.20, No.8, 1985, pp.82-87.

9. В.Ф.Турчин. Метаязык для формального описания алгоритмических языков. - В сб.: Цифровая вычислительная техника и программирование, М.: Сов. радио, 1966, с.116-124.

10. В.Ф.Турчин. Программирование на языке Рефал. Часть 5. Использование метафункций в языке Рефал. - Препринт / М.: Институт Прикладной Математики АН СССР, 1971, N 49, 55 с.

11. В.Ф.Турчин. Эквивалентные преобразования рекурсивных функций, описанных на языке Рефал. - В сб.: Теория языков и методы построения систем программирования. Труды симпозиума, Киев-Алушта: 1972, с. 31-42.

12. В.Ф.Турчин. Эквивалентные преобразования программ на Рефале. - В сб.: Автоматизированная система управления строительством. М.: ЦНИПИАСС, 1974, вып.4, с.36-68.

Климов Андрей Валентинович , Романенко Сергей Анатольевич "Метавычислитель для языка Рефал. Основные понятия и примеры."
Редактор В.С. Штракман. Корректор А.О. Лацис.

Подписано к печати 25.02.87г. № Т- 00254. Заказ № 173.
Формат бумаги 60X90 1/16. Тираж 280 экз.
055 (02)2 Объём 1,3 уч.-изд.л. Цена 15 коп.

Отпечатано на ротапринтах в Институте прикладной математики АН СССР

Москва, Миусская пл. 4.

Ссылки на издание рекомендуется делать по следующей форме: и.о., фамилия, название, препринт Ин. прикл. матем. им. М.В. Келдыша АН СССР, год, №.

Распространение: препринты института продаются в магазинах Академкниги г. Москвы, а также распространяются через Библиотеку АН СССР в порядке обмена.

Адрес: СССР, 125047, Москва-47, Миусская пл. 4, Институт прикладной математики им. М.В. Келдыша АН СССР, ОНТИ.

The references should be typed by the following form: initials, name, title, preprint, Inst.Appl.Mathem., the USSR Academy of Sciences, year, N(number).

Distribution. The preprints of the Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences are sold in the bookstores "Academkniga", Moscow and are distributed by the USSR Academy of Sciences Library as an exchange.

Adress: USSR, I25047, Moscow A-47, Miusskaya Sq.4, the Keldysh Institute of Applied Mathematics, Ac.of Sc., the USSR, Information Bureau.

Цена 15 коп.