МЕТОДЫ И ИНСТРУМЕНТЫ КОНСТРУИРОВАНИЯ ПРОГРАММ

Серия
“КОНСТРУИРОВАНИЕ И ОПТИМИЗАЦИЯ ПРОГРАММ”

Под редакцией
доктора физ.-мат. наук, профессора, чл.-корр. РАЕН
В. Н. Касьянова

Выпуски серии:

1. Смешанные вычисления и преобразование программ (1991)
2. Конструирование и оптимизация программ (1993)
3. Интеллектуализация и качество программного обеспечения (1994)
4. Проблемы конструирования эффективных и надежных программ (1995)
5. Оптимизирующая трансляция и конструирование программ (1997)
6. Проблемы систем информатики и программирования (1999)
7. Поддержка супервычислений и Интернет-ориентированные технологии (2001)
8. Касьянов В. Н., Мирзуитова И. Л. Slicing: срезы программ и их использование (2002)
9. Современные проблемы конструирования программ (2002)
10. Новые информационные технологии в науке и образовании (2003)
11. Программные средства и математические основы информатики (2004)
12. Методы и инструменты конструирования и оптимизации программ (2005)
13. Проблемы интеллектуализации и качества систем информатики (2006)
14. Касьянова Е. В. Адаптивные методы и средства поддержки дистанционного обучения программированию (2007)
15. ***Методы и инструменты конструирования программ***

Российская академия наук
Сибирское отделение
Институт систем информатики
имени А. П. Ершова

МЕТОДЫ И ИНСТРУМЕНТЫ КОНСТРУИРОВАНИЯ ПРОГРАММ

Под редакцией
проф. Виктора Николаевича Касьянова

Новосибирск 2007

УДК 519.68; 681.3.06
ББК З 22.183.49+ З 22.174.2

Методы и инструменты конструирования программ. — Новосибирск: Ин-т систем информатики имени А. П. Ершова СО РАН, 2007. — 235 с.

Является пятнадцатым в серии сборников, издаваемых Институтом систем информатики имени А.П.Ершова СО РАН. Описывает проблемы интеллектуализации и качества систем информатики.

Сборник представляет интерес для системных программистов, а также студентов и аспирантов, специализирующихся в области системного и теоретического программирования.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

TOOLS AND TECHNIQUES OF PROGRAMM CONSTRUCTION

Edited by
prof. V. N. Kasyanov

Novosibirsk 2007

This volume is the fifteenth one in a series of books published in A.P. Ershov Institute of Informatics Systems. This volume is devoted to the tools and techniques of program construction and optimization.

The volume is of interest for system programmers, students and post-graduates working in the field of system and theoretical programming.

ПРЕДИСЛОВИЕ РЕДАКТОРА

Пятнадцатый выпуск серии «Конструирование и оптимизация программ» посвящен решению актуальных проблем конструирования программ, главным образом методологии конструирования и оптимизации параллельных программ.

Продолжая уже сложившиеся традиции, данный выпуск, как и предыдущие, базируется на результатах исследований, ведущихся в лаборатории по конструированию и оптимизации программ Института систем информатики имени А.П. Ершова СО РАН совместно с кафедрой программирования Новосибирского государственного университета. Основная часть данного сборника написана по результатам первого года работ по проекту «Разработка и реализация интегрированной визуальной среды конструирования и оптимизации параллельных программ», выполняемого сотрудниками лаборатории при финансовой поддержке Российского фонда фундаментальных исследований (грант РФФИ № 07-07-12050).

В статье Арапбаева Р.Н. приведены результаты экспериментального исследования новой стратегии тестирования на стандартном наборе тестовых научных программ NASA и PERFECT Club benchmarks и наборе циклов, собранном из статей, посвященных вопросам анализа зависимостей по данным.

Статья Евстигнеева В.А. и Турсунбай кызы Ы. посвящена раскраске ω-совершенных графов в рамках распределенной модели вычислений, которая использует широко известную стратегию ПН-алгоритма.

В первой статье Идрисова Р. И. формулируются задачи распараллеливания программ в терминах внутреннего представления IR2 системы функционального программирования SFP. Вторая его статья посвящена рассмотрению основных методов межпроцедурного анализа с их ориентацией на использование при автоматическом распараллеливании программ.

В статье Касьянова В.Н. и Стасенко А.П. описывается синтаксис и семантика новой версии входного языка Sisal 3.2 системы функционального программирования SFP.

В первой статье Крайниковского С.С. рассматриваются вопросы создания графических интерфейсов и визуализации данных в геофизических программных системах. Вторая его статья посвящена вейвлет-обработке данных в геофизических исследованиях скважин и содержит обзор геофизических исследований скважин и методов обработки данных.

Статья Марчука П.А. содержит описание технологии поддержки распределенной фактографической информационной системы, создаваемой в рамках проекта «Электронный фотоархив СО РАН.

В статье Несговоровой Г.П. обсуждаются вопросы применения информационно-коммуникационных технологий в сохранении российского и мирового культурного наследия.

Статья Пыжова К.А. посвящена вопросам представления функциональных программ, ориентированных на их оптимизацию и эффективное распараллеливание вычислений.

В статье Стасенко А.П. вводится и исследуется магазинная автоматная модель, подходящая для наглядного описания эффективного нисходящего синтаксического разбора.

Статья Филябина С.В. содержит описание технологии автоматизации мониторинга и контроля легальности финансовых операций современных кредитных организаций.

В статье Шпака М.В. описаны некоторые математические постановки задач электрического и электромагнитного каротажа и приведен перечень основных методов их решения.

Проф. В.Н. Касьянов

Р. Н. Арапбаев

ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ НОВОЙ СТРАТЕГИИ ТЕСТИРОВАНИЯ*

К настоящему времени разработаны многие алгоритмы для анализа зависимостей по данным [1, 2]. В [3] выработана *новая стратегия* применения тестов для выявления зависимости по данным, в которой алгоритм стратегии состоит из серии эффективных и не дорогостоящих, имеющих линейную и полиномиальную сложность тестов на зависимость. В работе учтены результаты эмпирических и теоретических исследований анализа зависимостей по данным [4, 5], а также некоторые ограничения аналогичных работ [6–9]. В настоящей работе проведено экспериментальное сравнение результатов предложенного метода с наиболее известными стратегиями тестирования анализа зависимостей по данным, такими как Эпсилон-тест [7] и алгоритм Майдана [8]. Эксперимент проведен с использованием инструмента Petit V1.2 [10], разработанного в Мэриленд-ском университете как расширенный вариант инструмента tiny [11], и с использованием системы SUIF [12], разработанной в Стенфордском университете. Для эксперимента использованы два вида данных. Первый вид — набор тестовых научных программ NASA и PERFECT Club benchmarks [13], где каждая программа включает от 500 до 18000 строк. Второй вид — набор из 16 циклов, собранный из работ, аналогичных нашей [4, 6, 8, 9, 14, 15].

Все понятия, не определяемые в этой работе, могут быть найдены в [1, 2].

1. СТРАТЕГИЯ ТЕСТИРОВАНИЯ

При распараллеливании циклов одной из важных проблем является выявление зависимости по данным. Тесты на зависимость должны определять, существуют ли целочисленные решения следующей системы линейных диофантовых уравнений (1), полученной при выявлении зависимостей, удовлетворяющей ограничениям границ циклов (2):

* Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n + c_1 = 0\\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n + c_2 = 0\\
&\qquad\qquad \ldots\ldots\ldots\\
&a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n + c_m = 0
\end{aligned}
\qquad (1)
$$

и

$$L_i \le x_i \le U_i \quad \text{где } i=1,\ldots,n \qquad (2)$$

1.1. Алгоритм новой стратегии

В работе [3] предлагается новая стратегия применения тестов для выявления зависимости по данным, в которой алгоритм состоит из серии эффективных и недорогостоящих тестов на зависимость.

В данной стратегии, в зависимости от значений основных параметров задачи:

- размерность массивов;
- количество вложенных циклов;
- значения коэффициентов индексных переменных и значения границ циклов,

в первую очередь выделяются часто встречающие и легко разрешимые случаи. Соответственно каждому случаю применяется один быстрый и точный тест или серия эффективных тестов.

Итак, на вход нашего алгоритма подается гнездо циклов, в котором ***r*** — количество вложенных циклов, и операторы цикла обращаются к элементам ***d***-мерного массива. Кроме того, считаются постоянными и известными значения коэффициентов индексных переменных $a_{11}, a_{12}, \ldots, a_{mn}$ и значения границ циклов $L_1, L_2, \ldots, L_n, U_1, U_2, \ldots, U_n$, где $n = 2 * \boldsymbol{r}$ и $m = d$. Задача нашего алгоритма — выявить зависимости по данным между операторами в итерациях гнезда циклов, т.е. алгоритм должен возвращать ответ «да/нет» о существовании целочисленных решений $i_1, i_2 \ldots, i_n$ системы линейных диофантовых уравнений (1), удовлетворяющих ограничениям (2).

Для этого сначала выделены часто встречающие и легко разрешимые случаи задачи зависимости по данным:

1. ***r*= 1**, ***d* = 1**, т.е. внутри единственного цикла операторы обращаются к элементам одномерного массива. В этом случае уравнение зависимости (1) выглядит так: $a_1 x_1 + a_2 x_2 = a_0$ и $L \le x_1, x_2 \le U$. Для уравнения целесообразно применить самый быстрый и точный *SIV-тест* [6].

2. $\boldsymbol{r > 1}$, $\boldsymbol{d = 1}$, уравнение зависимости имеет вид: $a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = a_0$, где $L_i \le x_i \le U_i$, $i=1,\ldots,n$. Этот случай несколько усложняет решение, следовательно, применяется серия одномерных тестов на зависимость: тест Банержи [16], I-тест [17] и IR-тест [18]. Каждый следующий тест выполняется только в том случае, если был получен неточный ответ (maybe) предыдущим тестом, кроме того, после применения теста Банержи выполняется проверка коэффициентов индексных переменных для уточнения ответов теста.

3. $\boldsymbol{d = 2}$ и имеются *сцепленные индексы*. Система уравнений зависимости имеет вид:
   $a_{1,1} x_1 + a_{1,2} x_2 + \ldots + a_{1,n} x_n = a_{1,0}$
   $a_{2,1} x_1 + a_{2,2} x_2 + \ldots + a_{2,n} x_n = a_{2,0}$
   и $L_i \le x_i \le U_i$ где $i = 1, \ldots, n$.

   Этот случай доминирует в реальных последовательных программах, но применение обычных одномерных тестов на зависимость в этом случае бесполезно, так как имеются сцепленные индексные переменные. Поэтому применяется серия многомерных тестов: λ-тест [19], многомерный I-тест [20] и модифицированный λ-тест [21]. Метод запоминания результатов предыдущих тестов и использование их для последующих тестов оптимизирует данный случай.

4. В оставшихся случаях уравнение зависимости имеет вид (1) с ограничениями (2). Каждое уравнение рассматривается в отдельности, и для него последовательно применяется серия одномерных тестов: тест Банержи, I-тест и IR-тест. Этот подход дает более точный ответ, если индексные переменные *не сцеплены.* На практике доля сцепленных индексных переменных в ссылках трехмерных массивов и выше незначительна.

Учитывая все случаи, была собрана и реализована библиотека тестов на зависимость. Библиотека состоит из следующих тестов: ZIV-тест, SIV-тест, НОД-тест, Банержи-тест, I-тест, IR-тест, λ-тест, многомерный I-тест и модифицированный λ-тест. Кроме тестов на зависимость в библиотеке имеются алгоритмы для уточнения ответов теста Банержи (см. разд. 1.2.4). Все алгоритмы имеют линейную временную сложность. Из-за высокой стоимости в библиотеку не вошли точные тесты. На рис. 1 приведена общая схема новой стратегии применения тестов на зависимость.

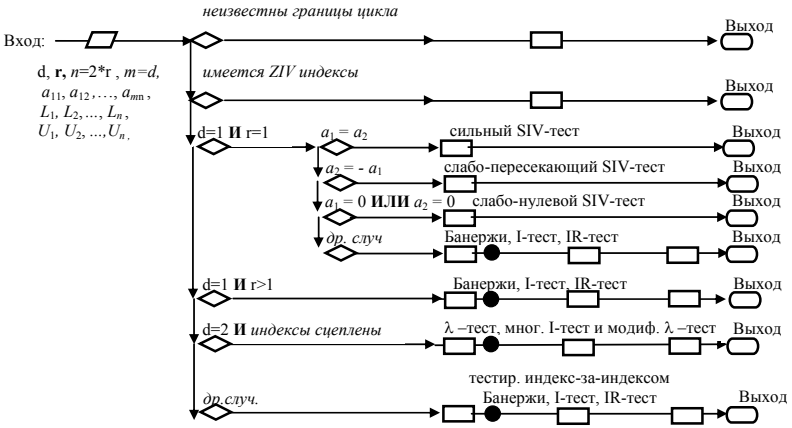


Рис. 1. Общая схема новой стратегии тестирования.

● — алгоритм проверки коэфц., □ — тесты на зависимость

Алгоритм новой стратегии может останавливаться в нескольких случаях, если был получен точный ответ любого теста или после применений серии одномерных тестов либо многомерных тестов на зависимость. Метод запоминания результатов предыдущих тестов и использование их для последующих тестов, а также применение алгоритмов уточняющих ответы теста Банержи, оптимизируют общее время выполнения алгоритма новой стратегии.

1.2. Построение новой стратегии тестирования

В этом подразделе будут показаны наиболее важные преимущества новой выработанной стратегии и некоторые требования к алгоритмам зависимостей по данным.

1.2.1. Аналогичные алгоритмы анализа зависимостей по данным

Исследование алгоритмов на зависимость выявило несколько работ, в некотором отношении аналогичных к нашей работе. Во всех исследованиях был собран набор тестов на зависимость с целью использования их для ре-

шения проблемы в практических ситуациях эффективно и точно. Однако наша работа по существу отличается от них.

Дельта-тест [6] разработан для определенных классов ссылок массива, которые часто встречаются в научных программных кодах. Тест сначала классифицирует индексные выражения массивов на следующие категории: ZIV (нулевая индексная переменная), SIV (единственная индексная переменная) и MIV (составная индексная переменная) формы. Соответственно к каждой форме применяется быстрые и точные одноименные тесты.

В работе [7] представлен более упрощенный и быстрый вариант Дельта-теста, называемый **Эпсилон-тестом**. В этом тесте рассмотрены только самые простые случаи индексных выражений SIV, не используются *сцепленные* MIV формы и НОД-тест, а также не рассматриваются треугольные границы цикла при использовании теста Банержи. Хотя эти алгоритмы являются самыми быстрыми, но они уступают по точности предложенному в данной работе алгоритму.

Алгоритм Майдана [8], который использован в системе SUIF Стенфордского университета, состоит из серии точных тестов, каждый из которых применим в ограниченной области. Последний тест в алгоритме — метод исключения Фурье—Моцкина, расширенный для решения целочисленных задач. Авторы показали, что практически зависимость по данным может быть вычислена точно и эффективно. Главное различие между алгоритмом Майдана и предложенным подходом — в том, что в первом случае добивались требуемого результата с использованием дорогих методов. В противоположность этому наш подход пытается получит те же результаты с использованием более дешевых тестов на зависимость.

K-тест [9] также состоит из библиотеки тестов на зависимость, но в отличие от других, вместо конкретной стратегии применения тестов используются методы искусственного интеллекта, хотя в самой работе также упоминается о NP-полноте методов искусственного интеллекта.

1.2.2. Основные результаты существующих эмпирических исследований

Отметим, что объектом автоматического распараллеливания служат большие научные пакеты прикладных программ, написанных на последовательных языках типа Фортран. Согласно эмпирическому изучению [4], в реальных программах индексные выражения не очень сложны. Из всех исследованных массивов примерно 56% составляют ссылки одномерных массивов и 36% — ссылки двумерных массивов. Доля ссылок трехмерных массивов и выше около 8 %. Что касается индексных выражений массивов, то 53 % являются линейными, 13 % — частично линейными и 34 % — нели-

нейными. Поэтому обычно для анализа зависимостей по данным на практике используются только одномерные тесты, использующие подход тестирования «индекс-за-индексом». В многомерных случаях система уравнений зависимости может не иметь решения даже в том случае, когда имеются решения в каждом из отдельных уравнений.

В новой стратегии для анализа ссылок одномерных массивов применяется серия из трех эффективных тестов: тест Банержи, I-тест и IR-тест.

При анализе многомерных массивов основную трудность вызывают часто встречающиеся в реальных программах *сцепленные* индексы. Как показано в эмпирических исследованиях З. Шена и др. [4] , более чем в девяти тысячах парах двумерных ссылок массивов приблизительно 46% являются сцепленными индексными выражениями. Что касается ссылок массивов большей размерности, то только 2% являются сцепленными индексными выражениями. Поэтому на практике важно иметь эффективный тест для обработки сцепленных индексов, особенно для анализа ссылок двухмерных массивов. Одним из таких эффективных алгоритмов является λ-тест.

Для обработки сцепленных индексов в ссылках двухмерных массивов, мы предлагаем применить серию многомерных тестов на зависимость: λ-тест, многомерный I-тест и модифицированный λ-тест. Что касается ссылок массива большей размерности, доля которых незначительна, то для них предлагается применять серии из одномерных тестов (тестирование “индекс-за-индексом”).

1.2.3. Появления новых тестов выявления зависимости по данным

К настоящему времени разработано множество тестов на зависимости, дающие приближенные и точные решения задачи выявления зависимости по данным. Далее кратко следует описание новых тестов на зависимость, применение которых при выработке новой стратегий дало ощутимые результаты при выявлении зависимости по данным на реальных задачах.

I–тест [17]

Данный тест является комбинацией тестов НОД и Банержи. Как и НОД тест, он проверяет существование целочисленного решения; как и тест Банержи, он учитывает ограничения индексных переменных. I-тест преобразует каждое уравнение зависимости (1) в интервальное уравнение:

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = [L, U] \qquad (3)$$

В правой части уравнения (3) L, U — верхняя и нижняя границы, являющиеся константами. В исходной форме верхняя и нижняя границы равны постоянному значению в правой части уравнения из системы (1). Тест в каждой итерации, выбирая переменную из левой части уравнения, перемещает ее в верхнюю и нижнюю границу в правой части (используя механизм *теста Банержи*), затем применяет НОД-тест к оставшимся коэффициентам. Этот процесс продолжается до тех пор, пока не будет доказано, что уравнение является не допустимым или нет больше переменных, которые могут быть перемещены. Подробное теоретическое объяснение I-теста и некоторые преобразования интервальных уравнений описаны в [1].

Многомерный I-тест [20]

I-тест является эффективным и более точным методом анализа зависимостей по данным для одномерных массивов по сравнению с тестом Банержи. При анализе многомерных массивов λ-тест является эффективным тестом, но λ-тест дает ответ о существование вещественных решений системы уравнений зависимости. Многомерный I-тест представляет собой комбинацию данных двух тестов, следовательно, он дает более точный ответ о существовании целочисленных значений решений системы.

λ-тест

λ-тест [19] предназначен для *сцепленных* и *многомерных* ссылок массивов. Тест решает систему уравнений (1) и неравенств (2) и определяет, имеет ли данная система действительные решения.

Геометрически каждое линейное уравнение в (1) представляет собой гиперплоскость π в пространстве $\mathbf{R^n}$. Пересечение **m** гиперплоскостей — $\boldsymbol{S}$ соответствует общим решениям системы (1). Очевидно, если $\boldsymbol{S}$ пусто, то никакой зависимости по данным нет. Границы циклов (2) соответствуют ограниченному выпуклому множеству **V** в $\mathbf{R^n}$. Уравнение имеет действительное решение, удовлетворяющее границам циклов и направлениям зависимостей, тогда и только тогда, когда соответствующая уравнению гиперплоскость π пересекается с **V**. Тестирование «индекс-за-индексом» определяет, пересекается ли каждая гиперплоскость π с **V**. Необходимо определить, пересекается ли **S** с **V**. Если из всех гиперплоскостей найдется такая гиперплоскость, которая не пересекает **V**, то очевидно **S** не может пересекаться с **V**. Однако, даже если каждая гиперплоскость из (1) пересекает **V**, существует вероятность того, что **S** не пересечет **V**. Если можно найти новую гиперплоскость, которая содержит **S**, но не пересекает **V**, то это дока-

зывает, что **S** и **V** не пересекаются. Имеется бесконечное число таких гиперплоскостей. Задача λ-теста — исследовать по мере необходимости некоторое количество гиперплоскостей, для определения пересечения **S** и **V**. В общем случае λ-тест генерирует C_n^{m-1} таких гиперплоскостей, которые являются линейными комбинациями уравнений (1) и называются λ-плоскостями. Чтобы определить, пересекается ли каждая λ-плоскость с **V**, применяется тест Банержи для каждой λ-плоскости. Если хотя бы одна из λ-плоскостей не пересекает **V**, тогда зависимости по данным нет. Если каждая λ-плоскость пересекает **V**, то λ-тест принимает решение о возможном существовании зависимости.

IR-тест

IR-тест [20] является точным тестом, но он неприменим, когда значения границ циклов неизвестны или не являются константами. IR-тест находит целочисленные решения уравнения зависимости путем сокращения интервала решений переменных с многократным проектированием. Как только эффективный интервал решений какой-нибудь переменной сжимается к пустому, то уравнение зависимости, содержащее данную переменную, не имеет целочисленного решения.

Модифицированный λ-тест

В *модифицированном λ-тесте* [21] λ-тест интегрирован с точным **IR**-тестом, благодаря чему при анализе зависимостей многомерных массивов были получены более точные результаты.

Модифицированный λ-тест с помощью λ-теста генерирует множество линейных комбинаций гиперплоскостей (см. λ-тест). К сгенерированным гиперплоскостям применяется **IR**-тест для нахождения гиперплоскости из данного множества, которая не имеет целочисленных точек пересечения с **V**.

Идея нашей стратегии опирается на следующие научные факты и результаты.

1.2.4. Случаи, повышающие точность тестов на зависимость

Точно определяющие методы: Омега-тест, Power-тест, алгоритм Майдана и др. используют линейные и целочисленные методы для решения диофантовых уравнений, например, метод Фурье—Моцкина, Симплекс метод и др., которые не эффективны на практике. В экспериментальных результатах Р. Триоле [22] показано, что по сравнению с более простыми

методами метод исключения переменных Фурье—Моцкина выполняется в 22–28 раз дольше.

Одним из стандартных и распространенных тестов на зависимость является тест Банержи. Он является приближенным тестом и принимает во внимание границы циклов. Эффективность и полноценность теста Банержи при опровержении зависимостей делают его одним из самых используемых тестов в распараллеливающих компиляторах

В исследованиях [16, 19, 5] показано, что если коэффициенты линейного уравнения удовлетворяют некоторым условиям, то тест Банержи становится точным тестом. Банержи показал, что его неравенства точны, если все коэффициенты индексных переменных равны 1, 0, или -1 [16]. Ли и др. [19] показали, что неравенства Банержи точны, если коэффициент одной индексной переменной $|a_k|=1$ и $|a_i| \le (U_i - L_i)$, где $i=1,\ldots,k-1,k+1,\ldots,n$.

Клапфолз (Klappholz) и др. [5] доказали, что неравенства Банержи точны, если после упорядочения коэффициентов индексных переменных $|a_1| \le |a_2| \le \ldots \le |a_n|$, коэффициент индексной переменной $|a_1| = 1$, и для каждого j выполняется следующее условие: $\left|a_j\right| \le 1 + \sum_{k=1}^{j-1} \left|a_k\right| (U_k - L_k)$, $2 \le j \le n$.

2. ЭКСПЕРИМЕНТАЛЬНОЕ СРАВНЕНИЕ РЕЗУЛЬТАТОВ

Данный раздел посвящен анализу экспериментальных результатов, подтверждающих эффективность и корректность новой стратегии тестирования.

2.1. Системная среда

Эксперимент проведен с использованием инструмента **Petit V1.2** [10], разработанного в Мэрилендском университете как расширенный вариант инструмента ***tiny*** [11] и с использованием системы **SUIF** [12], разработанной в Стенфордском университете.

Petit является исследовательским инструментом реструктурирования программ. Он поддерживает совокупность библиотек и фундаментальных операций для анализа зависимостей по данным и реструктурирования программ. В нем в качестве алгоритмов анализа зависимостей по данным вне-

дрены четыре теста: Омега-тест, тест Банержи, Эпсилон-тест и Омега-Эпсилон-тест.

SUIF представляет собой инфраструктуру для исследований в области распараллеливающих и оптимизирующих компиляторов. Как сказано выше, в качестве библиотеки зависимости в системе служит алгоритм Майдана.

Обе программы были установлены на персональном компьютере с процессором AMD Athlon XP 1700+ с операционной системой Debian GNU Linux. Для эксперимента использованы два вида данных. Первый — набор тестовых научных программ NASA и PERFECT Club benchmarks [13], где каждая программа включает от 500 до 18000 строк (см. Таблицу 1). Второй вид — набор из 16 циклов, собранный из работ, аналогичных нашей [4, 6, 8, 9, 14, 15]. Все циклы являются специальными примерами и созданы для демонстрации мощности некоторых тестов на зависимость по данным.

Таблица 1

Статистические характеристики эталонных тестовых программ

№	Эталонные тестовые программы	Количество строк программ	Количество подпрограмм	Количество DO-циклов	Количество ссылок на массивы
1	**LGSI**	2815	36	161	6389
2	**LWSI**	1430	17	56	906
3	**SDSI**	8446	80	259	658
4	**TISI**	579	8	78	84
5	**btrix**	159	1	14	488
6	**cholsky**	53	1	18	70
7	**gmtry**	117	1	14	369
8	**gosser**	19	2	5	7
	Всего	**13618**	**146**	**605**	**8971**

В табл. 1 приведены статистические характеристики эталонных тестовых программ. Отметим, что эти программы специально были собраны для тестирования методов разных распараллеливающих и оптимизирующих компиляторов. Так как объектом распараллеливания служат большие научные пакеты прикладных программ, написанных на последовательных языках типа Фортран, каждая из этих программ решает задачу прикладной физики, математики, химии и др. В столбцах таблицы отражены названия, количество строк, количество подпрограмм, количество DO-циклов (for-циклы) каждой программы. При распараллеливании последовательных про-

грамм главным источником потенциального параллелизма, как правило, служит гнездо DO-циклов. В последнем столбце представлено количество ссылок на массивы.

2.2. Сравнение результатов

С помощью пакетов basesuif 1.3.0.1, suifbuilder 1.3.0.1, baseparsuif 1.3.0.1 и suifcookbook 1.3.0.1 системы SUIF [12] собраны два анализатора зависимостей по данным. Соответственно первый основан на алгоритме Майдана, а на втором внедрена новая стратегия. Каждый анализатор принимает на входе преобразованный на SUIF формат (с помощью scc драйвера) *.spd файл последовательной программы, а на выходе дает информацию о всех зависимостях по данным в данной программе. При экспериментальном сравнении результатов рассматривались только *истинные (потоковые) циклически порожденные зависимости по данным*.

Таблица 2

Сравнение результатов

№	Эталонные тестовые программы	Всего обрщ. на истин. завис.	Кол-во разрушенных зависимостей			
			Алгоритм Майдана		Новая стратегия	
1	**LGSI**	6168	5546	89,92%	5546	89,92%
2	**LWSI**	795	102	12,83%	102	12,83%
3	**SDSI**	417	154	36,93%	149	35,73%
4	**TISI**	52	0	0,00%	0	0,00%
5	**btrix**	450	208	46,22%	208	46,22%
6	**cholsky**	61	3	4,92%	0	0,00%
7	**gmtry**	258	125	48,45%	119	46,12%
8	**gosser**	5	0	0,00%	0	0,00%
	Всего	**8206**	**6138**	**74,80%**	**6124**	**74,63%**

Результаты каждого анализатора зависимостей по данным для эталонных тестовых программ показаны в табл. 2. Третья колонка табл. 2 представляет общее количество обращений на предмет наличия потоковой зависимости по данным, здесь тесты должны разрушать зависимости, если самом деле не существует потоковой зависимости по данным. Следовательно, в следующих колонках таблицы показано, на сколько процентов удалось

разрушить зависимости с помощью алгоритма Майдана и с алгоритма новой стратегии соответственно.

Алгоритм Майдана считается дорогим и точным тестом, т.к. он использует метод исключения Фурье—Моцкина, имеющий теоретически экспоненциальную временную сложность. Из табл. 2. видно, что результаты нового алгоритма очень близки к результатам точных тестов, хотя предложенный алгоритм имеет полиномиальную временную сложность в наихудшем случае.

2.3. Сравнение результатов на экспериментальных примерах

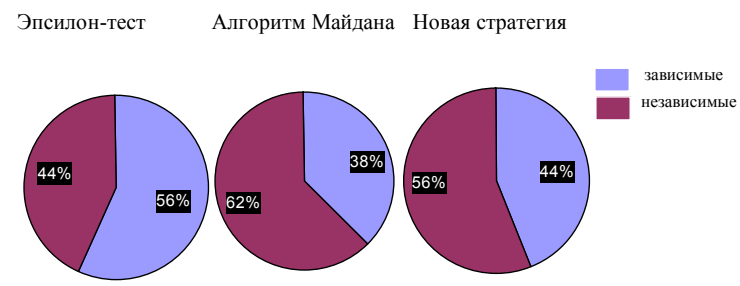
Для повышения качества эксперимента был собран набор из 16 циклов из работ аналогичных нашей: Maydan [8], Wolfe [14], Goff [6], Pugh [15], Yang [9] и Shen [4]. Все циклы являются специальными примерами и созданы для демонстрации мощности некоторых тестов на зависимость по данным. В данных примерах индексные выражения более сложны, чем реальные программы. Статистические данные показаны в табл. 3.

Таблица 3

Характеристики циклов в экспериментальных примерах

Пример / количество	1-мерные	2-мерные	**Всего**
Maydan	4	1	5
Wolfe	0	4	4
Goff	0	1	1
Pugh	1	0	1
Yang	2	1	3
Shen	2	0	2
Всего	9	7	**16**

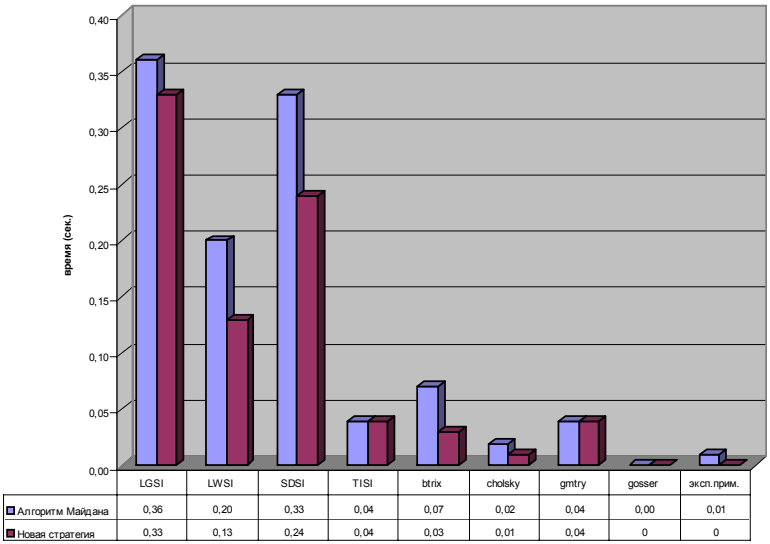
Общие статистические характеристики экспериментальных примеров: количество строк — 66, количество DO-циклов — 26, количество ссылок на массивы — 40. В этом случае мы сравнивали результаты следующих алгоритмов: Эпсилон-тест, алгоритм Майдана и новая стратегия тестирования. С помощью инструмента Petit к экспериментальным примерам применялся Эпсилон-тест. Соответственно для 40 пар ссылок массивов получены следующие данные о зависимости по данным (см. рис 2).



Всего ссылок	Эпсилон тест	Алгоритм Майдана	Новая стратегия
16	7	10	9

Рис. 2. Сравнение результатов на экспериментальных примерах

2.4. Время выполнения



	LGSI	LWSI	SDSI	TISI	btrix	cholsky	gmtry	gosser	эксп.прим.
Алгоритм Майдана	0,36	0,20	0,33	0,04	0,07	0,02	0,04	0,00	0,01
Новая стратегия	0,33	0,13	0,24	0,04	0,03	0,01	0,04	0	0

Рис. 3. Время выполнения

Для определения того, какой метод требует больше времени исполнения, использован *GNU* профилятор ***`gprof'*** [23] операционной системы Linux с периодом отсчета 0,01 сек. Чтобы снизить статистические неточности, каждый алгоритм зависимости по данным выполнен 100 раз для различных эталонных тестов, и взято их усредненное значение (см. рис. 3). В общем случае алгоритм Майдана требовал времени на 25% больше, чем алгоритм новой стратегии.

2.5. Статистические данные *Новой стратегии*

По итогам эксперимента были анализированы статистические данные новой стратегии (см. табл. 4). Результаты еще раз показывают целесообразность и эффективность методов, предложенных в предыдущей главе. В большинстве случаев зависимости по данным выявляются с помощью простых тестов: константный тест (ZIV-тест), SIV-тест, Банержи тест и λ-тест. Помощь более сложных тестов (I-тест, IR-тест, Многомерный I-тест и Модифицированный λ-тест) требовалась в незначительных случаях. Это достигается с помощью алгоритмов, уточняющих ответы теста Банержи. Так, в 361 случае с помощью Банержи теста в 54 случаях опровергнуты зависимости по данным, а в 301 случае алгоритм проверки коэффициентов доказывает существование зависимости. Это позволяет сократить общее время выполнения новой стратегии, так как после положительного ответа алгоритма проверки исключается выполнение последовательности более сложных тестов (I-тест, IR-тест или Многомерный I-тест, Модифицированный λ-тест). В 71 случае с помощью λ-теста опровергнуты зависимости по данным в 2 случаях, а 67 случае алгоритм проверки коэффициентов доказывает существование зависимости.

3. ЗАКЛЮЧЕНИЕ

Анализ зависимостей по данным является фундаментальным компонентом в распараллеливающих компиляторах. В данной работе предложен новый эффективный алгоритм анализа зависимостей по данным, а также проведено экспериментальное сравнение результатов предложенного метода с наиболее известными стратегиями тестирования анализа зависимостей по данным, такими как Эпсилон-тест и алгоритм Майдана.

Таблица 4

Статистические данные Новой стратегии

тесты на зависимость		LGSI	LWSI	SDSI	TISI	btrix	cholsky	gmtry	gosser	прак.дан.	Всего
Константный-тест	обращ.	5862	42	185	0	208	2	230	0	0	**6529**
	независ.	5455	18	149	0	208	0	119	0	0	**5949**
НОД-тест	обращ	4	163	82	20	198	56	19	5	2	**549**
	незав.	0	0	0	0	0	0	0	0	1	**1**
SIV-тест	обращ	266	100	11	0	0	0	9	0	5	**391**
	незав.	91	31	0	0	0	0	0	0	4	**126**
Тест Банержи	обращ	1	319	2	0	30	0	1	0	8	**361**
	незав.	0	53	0	0	0	0	0	0	1	**54**
	проверка	1	266	2	0	30	0	1	0	1	**301**
I-тест	обращ	0	0	0	0	0	0	0	0	8	**8**
	незав.	0	0	0	0	0	0	0	0	0	**0**
IR-тест	обращ	0	0	0	0	0	0	0	0	1	**1**
	незав.	0	0	0	0	0	0	0	0	1	**1**
Лямбда-тест	обращ	0	0	64	0	0	0	0	0	7	**71**
	незав.	0	0	0	0	0	0	0	0	2	**2**
	проверка	0	0	64	0	0	0	0	0	3	**67**
Многомерный I-тест	обращ	0	0	0	0	0	0	0	0	2	**2**
	незав.	0	0	0	0	0	0	0	0	0	**0**
Модифицированный Лямбда-тест	обращ	0	0	0	0	0	0	0	0	2	**2**
	незав.	0	0	0	0	0	0	0	0	1	**1**

Экспериментальное исследование показывает, что при анализе зависимостей по данным на эталонных тестовых программах NASA и PERFECT Club benchmarks, результаты нового алгоритма очень близки к результатам алгоритма Майдана. Хотя алгоритм Майдана считается дорогим и точным тестом, а предложенный новый алгоритм имеет полиномиальную вре-

менную сложность в наихудшем случае. В общем случае алгоритм Майдана требовал времени на 25% больше, чем алгоритм новой стратегии.

Сравнение результатов на экспериментальных примерах показало, что новый алгоритм выявляет примерно на 12% больше ложных зависимостей, чем аналогичные приближенные алгоритмы (Эпсилон-тест), и только примерно на 6% уступает алгоритму Майдана.

Все результаты, полученные экспериментальным путем, еще раз показывают целесообразность и эффективность новой стратегии тестирования, предложенной в данной работе.

Практическим результатом данной работы является алгоритм, который может быть использован в блоке анализа зависимостей по данным в проектируемой системе быстрого прототипирования компилятора.

СПИСОК ЛИТЕРАТУРЫ

1. **Евстигнеев В.А.** Анализ зависимостей: состояние проблемы // Системная информатика: Сб. науч. тр. / Ин-т систем информатики СО РАН. — Новосибирск: Наука, 2000. — Вып. 7. — С. 112–173.
2. **Евстигнеев В.А., Арапбаев Р.Н., Осмонов Р.А.** Анализ зависимостей: основные тесты на зависимость по данным // Сиб. журн. вычисл. математики / РАН. Сиб. отд-ние. — Новосибирск, 2007. — Т. 10, № 3. — С. 247–265.
3. **Арапбаев Р.Н., Осмонов Р.А.** Анализ зависимостей: новая стратегия тестирования // Тр. Междунар. конф. «Параллельные вычислительные технологии (ПаВТ'2007)». — Челябинск: Ид-во ЮУрГУ, 2007. — Т. 2. — С. 16–27.
4. **Shen, Z., Li, Z., Yew, P.-C.** An empirical study of Fortran programs for parallelizing compilers // IEEE Transaction on Parallel and Distributed Systems. — 1992. — Vol. 1 (1). — P. 356–364.
5. **Psarris K.** Program analysis techniques for transforming programs for parallel execution // Parallel Computing. — 2002. — Vol. 28. — P. 455–469.
6. **Goff G., Kennedy K., Tseng C.** Practical Dependence Testing // Proc. of the ACM SIGPLAN 91 Conf on Programming Language Design and Implementation, June 1991. — P. 15–29.
7. **Pugh W., Speismean T**. On Fast Array Data Dependence Tests. — Univ. of Maryland, College Park, January 3, 1999. — http://citeseer.ist.psu.edu/43683.htm
8. **Maydan D., Hennessy J., Lam M.** Efficient and Exact Data Dependence Analysis // Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation, June 1991. — P. 1–14.
9. **Yang C.-T., Tseng S.-S., Shih W.-C.** The K Test: an Exact and Efficient Knowledge-based Data Dependence Testing Method for Parallelizing Compilers // Proc. Natl. Sci. Counc. ROC(A). — 2000. — Vol. 24, N 5. — P. 362–372.

10. **Kelly W., Maslov V., Pugh W., et al.** Petit: a tool for analyzing and transforming array calculations. — Dept. of Computer Science, University of Maryland, College Park, April 1996. — http://www.cs.umd.edu/projects/omega/index.html
11. **Wolfe M.** The Tiny Loop Restructuring Research Tool // Proc. of the 1991 Internat. Conf. on Parallel Processing, St Charles, IL, August 1991.
12. **Wilson R.P., French R.S., Wilson C.S. a.o.** SUIF: An infrastructure for research on parallelizing and optimizing compilers // SIGPLAN Not. —1994. — Vol. 29, N 12. — P. 31–37.
13. **Berry M. et al.** The PERFECT Club benchmarks: effective performance evaluation of supercomputers. Technical Report UIUCSRD Rep. No. 827, University of Illinois Urbana-Champaign, 1989, 48 p.
14. **Wolfe M., Tseng C.** The Power Test for Data Dependence // IEEE Transactions on Parallel and Distributed Systems. September 1992.
15. **Pugh W.** The Omega test: a fast and practical integer programming algorithm for dependence analysis // Communs. of the ACM. — 1992. — Vol. 35(8). — P.102–114.
16. **Banerjee U.** Data dependence in ordinary programs. — Urbana, 1976. — (Univ.III., Technical Rep. 76-837).
17. **Kong X., Klappholz D., Pssaris K.** The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization // IEEE Transactions on Parallel and Distributed Systems. — 1991. — Vol. 2(1). — P. 342–349.
18. **Huang T.-C., Yang C.-M.** Data dependence analysis for array references // J. of Systems and Software. — 2000. — Vol. 52. — P. 55–65.
19. **Li Z., Yew P.-C., Zhu C.-Q.** An efficient data dependence analysis for parallelizing compilers. // IEEE Transaction on Parallel and Distributed Systems. — 1990. — Vol. 1(1). — P. 26–34.
20. **Chang W.-L., Chu C.-P., Wu J.** A multi-dimensional version of the I test // Parallel Computing. — 2001. — Vol. 27. — P. 1783–1799.
21. **Арапбаев Р.Н., Осмонов Р.А.** Анализ зависимостей по данным для многомерных массивов на базе модифицированного λ-теста // Проблемы интеллектуализации качества систем информатики. — Новосибирск: ИСИ СО РАН, 2006. — С. 7–23.
22. **Triolet R.,** Interprocedural analysis for program restructuring with PARAFRASE. — Urbana, 1985 — (Tech. Rep. / Univ. III. CSRD; N 538).
23. **Fenlason and Stallman**, GNU gprof, The GNU Profiler. — http:// www.gnu.org/manual/gprof-2.9.1/html_chapter/gprof_toc.html

В.А. Евстигнеев, Ы. Турсунбай кызы

ДИНАМИЧЕСКИЙ РАСПРЕДЕЛЕННЫЙ ПН-АЛГОРИТМ ДЛЯ РАСКРАСКИ w -СОВЕРШЕННЫХ ГРАФОВ

Данная работа посвящена раскраске w-совершенных графов в рамках распределенной модели вычислений, которая использует широко известную стратегию ПН-алгоритма. Класс w-совершенных графов довольно широкий и содержит в себе такие практически интересные классы графов, как, например, класс хордальных графов [6], который является одним из наиболее изученных и широко применяемых классов графов.

ВВЕДЕНИЕ

Рассмотрим задачу раскраски вершин графа в рамках распределенной модели вычислений. Распределенные вычисления на графах представляют собой такую организацию вычислений, при которой отсутствует всякая возможность использовать глобальные операции и механизмы, а также возможность получать информацию иначе, нежели используя информацию из локальной памяти соседей. Это означает, что в основе любого распределенного вычисления на графах лежит организация взаимодействия с соседними вершинами и пересылка локальной информации.

Область применения распределенных вычислений на графах — организация целенаправленной деятельности коллектива исполнителей (автономных устройств, ЭВМ в составе сети, распределенной вычислительной системы и т.п.) путем обмена сообщениями между «близкими» в некотором смысле членами коллектива и при отсутствии каких-либо глобальных механизмов.

Данная распределенная модель раскраски может быть использована в распределенных беспроводных сетях для устранения столкновений пакетов путем назначения ортогональных кодов радиостанциям [1].

Заметим, что не всегда легко добиться и скорости, и эффективности алгоритма. В работе [2] представлен распределенный алгоритм для раскраски графа в $(\Delta+1)$ цветов, где Δ — наибольшая степень вершины в графе. Время работы алгоритма $O(\log n)$. Будем называть этот алгоритм тривиальным. Данный алгоритм достаточно простой и быстрый, но не оптимальный. Действительно, количество цветов, используемых алгоритмом, близок к Δ, да-

же если граф двудольный. Неудивительно, что тривиальный алгоритм не имеет механизма экономии цветов. Дальнейшее усовершенствование тривиального алгоритма предложен в [3]. В данной работе приведен новый алгоритм для раскраски в O (Δ / log Δ) цвета, но этот алгоритм работает только на графах без триангуляторов.

Одним из способов улучшения выполнения распределенного алгоритма является представление стратегии раскраски в алгоритм, который, как известно, является эффективным в нераспределенных алгоритмах.

В работе [4] представлен распределенный алгоритм для раскраски вершин графа, который походит на стратегию Наибольшие Первые. Раскраска, полученная с помощью данного алгоритма оптимальна или близка к оптимальному для некоторых классов графов, таких как, полные k-сторонние, гусеницы, короны и двусторонние колеса.

В настоящей работе исследуется задача раскраски w-совершенных графов с помощью ПН-алгоритма, представлен динамический распределенный алгоритм для решения данной задачи.

1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Раскраской вершин графа G называется такое приписывание цветов его вершинам, что никакие две смежные вершины не получают одинаковые цвета. *Хроматическое число* χ (G) графа G определяется как наименьшее количество цветов, необходимых для раскраски G, а раскраска G χ (G) цветами называется *оптимальной раскраской* графа G. Задача раскраски графа заключается в нахождении оптимальной раскраски. Характерной особенностью этих задач является существование объектов, которые по каким-либо причинам не могут быть объединены в одну группу.

Поскольку задача определения хроматического числа принадлежит классу полиномиально полных задач, исследования в этой области ведутся в разных направлениях. Принципиальные трудности, которые возникают при раскраске графа и нахождении его хроматического числа, вынуждают, во-первых, найти и исследовать практически интересные классы графов, для которых задача раскраски полиномиально разрешима, и, во-вторых, вычислить или оценить хроматическое число графа с помощью других, более легко вычисляемых характеристик графа.

Одной из важных характеристик, связанных с хроматическим числом, является число Визинга—Вилфа $w(G) = \max_{G' \subseteq G} \delta(G') + 1$, где

$\delta(G') = \min\limits_{x \in V(G')} d_{G'}(x)$ — минимальная степень графа G', а $d_{G'}(x)$ — степень вершины x в G'. $w(G)$ в качестве верхней оценки хроматического числа впервые рассмотрена в работе Секереша и Вилфа [5].

Важность этой характеристики заключается в том, что, во-первых, $w(G)$ является довольно нетривиальной верхней оценкой для $\chi(G)$ [5], т.е. класс графов, для которых $w(G) = \chi(G)$, довольно большой и содержит в себе много практически интересных классов, и, во-вторых, она легко вычисляемая.

Определение. Граф, обладающий тем свойством, что хроматическое число и вырожденность (число Визинга—Вилфа) равны не только у самого графа, но и у каждого его порожденного подграфа называется w-*совершенным графом*.

Важным подклассом w-совершенных графов являются хордальные графы [6].

Определение. Граф называется *хордальным (триангулированным)*, если каждый его цикл длины > 3 содержит хорду, т.е. ребро, соединяющее несмежные вершины простого цикла.

Более подробные определения и свойства w-совершенных графов приведены в работе [7, 8].

Для вычисления $w(G)$ вводится понятие упорядочения по наименьшему последнему (*ПН-упорядочение*) [9, 10, 11] графа G. Оно строится следующим образом:

а) для $n = n(G)$ в качестве v_i выбирается вершина минимальной степени в графе G;

б) для $i = 2,3,...,n$ в качестве v_i выбирается вершина минимальной степени в подграфе $G \setminus \{v_1,...,v_{i-1}\}$.

Для упорядоченного множества вершин $v_1,...,v_n$ графа G последовательной раскраской, отвечающей этому порядку, называется раскраска, определяемая следующим образом:

а) вершине v_n приписан цвет 1;

б) для $i = n-1,...,1$ вершина v_i получает цвет с наименьшим номером, не встречающийся на смежных с вершиной v_i вершинах.

Процедура определения ПН-упорядочения вершин и нахождения по нему раскраски называется *ПН-алгоритмом* [9]. По своему строению ПН-алгоритм приводит к раскраске не более чем $w(G)$ цветами.

2. ОПИСАНИЕ АЛГОРИТМА

Определение. *Параллельно-последовательным вычислительным алгоритмом* называется локальный алгоритм, каждый шаг которого состоит в обработке параллельно и независимо всех (или некоторых) вершин, смежных с вершинами, обработанными на предыдущем шаге; на первом шаге обрабатывается фиксированное множество начальных вершин [12].

Теорема. Задача раскраски w-совершенных графов ПН-алгоритмом разрешима в классе распределенных параллельно последовательных алгоритмов.

Доказательство:
Рассмотрим описание соответствующего распределенного алгоритма.
Мы предположим, что система синхронизирована в раундах.
Каждая вершина имеет следующие параметры:

- случайное значение: ***rndvalue (v);***
- номер, соответствующий ПН-упорядочению: ***SLnumber (v)*** (в начале номера всех вершин равны единице, т.е. SLnumber (v)=1);
- показатель состояния параметра SL-number: ***cond (v),*** который имеет либо значение I (intermediate) — промежуточное, либо значение F (final) — конечное (в начале все вершины имеют промежуточное состояние);
- количество соседних вершин, для которых не установлены конечные номера, т.е. cond (v) = I: ***ddeg (v)*** (в начале ddeg (v) = deg (v));
- палитра «запрещенных» цветов, цвета, которые были использованы соседними вершинами: ***usedcolor (v)*** (в начале пустая).

Пусть $v_1, v_2 \in V$. Мы говорим, что вершина v_1 имеет более высокий приоритет чем v_2, если: ddeg (v_1) < ddeg (v_2) или (ddeg (v_1) = ddeg (v_2)) и (rndvalue (v_1) < rndvalue (v_2)).

В каждом раунде все неокрашенные вершины параллельно и независимо друг от друга проделывают следующее:

1. Вершина v выбирает параметр rndvalue (v) $\in$ [0..1];

2. Посылает всем соседям следующие параметры: ddeg (v), rndvalue (v);

3. Сравнивает свои параметры с полученными от соседей и проверяет, какая вершина имеет более высокий приоритет;

4. Если вершина v имеет высокий приоритет, то она оставляет себе текущее значение параметра SLnumber и меняет значение параметра cond (v) с промежуточного на конечный. В противном случае, увеличивает на единицу значение параметра SLnumber (v);

5. Пересчитывает параметр ddeg (v).

6. Если ddeg (v) = 0, т.е. всем смежным вершинам назначены конечные ПН-номера, увеличивает на единицу значение параметра SLnumber и меняет значение параметра cond (v) с промежуточного на конечный, переход к шагу 7, иначе переход к шагу 2;

7. Посылает всем соседям параметры: SLnumber (v) и первый предполагаемый цвет с наименьшим номером (не находящийся в списке «запрещенных»);

8. Сравнивает свои параметры с полученными от соседей, проверяет, какая вершина имеет наибольший SLnumber, если вершина имеет наибольший номер, то оставляет предполагаемый цвет и стоп.

9. В противном случае обновляет список usedcolor (v), переход к шагу 6.

ПН-алгоритм можно условно разделить на два этапа:

1. Каждой вершине, имеющей минимальную степень, устанавливается номер, соответствующий ПН — упорядочению (1–6 шаг);
2. Производится раскраска графа G, начиная с вершин, которые имеют большие ПН-номера (7–9 шаг).

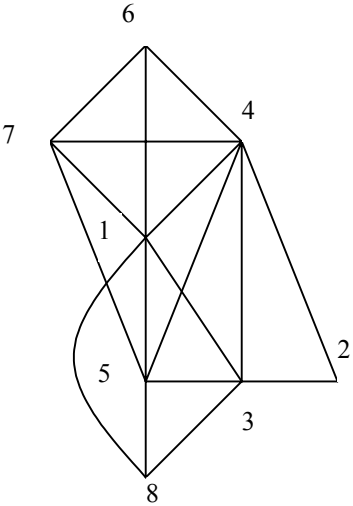


Рис. 1. w -совершенный граф

Рассмотрим пример, показанный на рис. 1.

В начале всех раундов граф находится в следующем состоянии: каждая вершина имеет параметр ddeg (v), значением которого является степень данной вершины; SL номера всех

вершин равны 1; состояния данных параметров является промежуточными; ни одна вершина не окрашена, и список «запрещенных» цветов пуст.

В начале алгоритма все вершины выбирают себе значение параметра rndvalue из интервала [0..1]. В первом раунде конечное состояние параметра SLnumber примут вершины 2 (ddeg (2) = 2), 6 (ddeg (2) = 3) и 8 (ddeg (2) = 3), которые имеют наименьшие степени в своей окрестности и соответственно имеют высокий приоритет. Данные вершины не являются соседними и «не влияют» друг на друга, поэтому все они сохраняют начальное значение параметра SLnumber (v) = 1. Остальные вершины увеличат на единицу значение данного параметра и пересчитывают значение ddeg (v).

Во втором раунде SL номера получат вершины 3 (ddeg (2) = 3) и 7(ddeg (2) = 3).

В третьем раунде смежные вершины 1, 4 и 5 имеют одинаковые параметры ddeg и в данном случае приоритет той или иной вершины определяет случайное значение rndvalue. В этом раунде конечный SLnumber получит вершина 1 (rndvalue = 1.09, ddeg (4) = 2).

В четвертом раунде конечный SLnumber будет назначен вершине 4 (rndvalue = 1.38б ddeg (1) = 1) и в пятом раунде вершине 5 (rndvalue = 1.62, ddeg (5) = 0).

После окончания каждого раунда для каждой вершины проверяется условие: ddeg (v) = 0. Если ответ положительный, тогда вершина переходит ко второму этапу алгоритма, в противном случае продолжает выполнять операции первого этапа.

На втором этапе вершина, для которой ddeg (v) = 0 отсылает всем соседям свой SLnumber и первый предполагаемый цвет с наименьшим номером. Необходимо отметить, что вершина не может быть окрашена в предполагаемый цвет до тех пор, пока данное условие не будет выполнено для всех вершин окрестности 2го порядка.

В 6-м раунде вершина 5, которая имеет наибольший SLnumber среди соседних вершин, будет окрашена в предполагаемый цвет с номером 1. Данная вершина останавливает все свои действия. Остальные вершины обновляют список «запрешенных» цветов usedcolor. Неокрашенные вершины продолжают аналогичные действия. После 10го раунда распределение цветов будет выглядеть следующим образом: вершина 4 (цвет — 2, раунд — 7), вершина 1 (цвет — 3, раунд — 8), вершины 3, 7 (цвет — 4, раунд — 9), вершины 2, 6 (цвет — 1, раунд — 10), вершина 1 (цвет — 2, раунд — 10).

СПИСОК ЛИТЕРАТУРЫ

1. **Battiti R., Bertossi A.A., Bonuccelli M.A.** Assigning codes in wireless networks // Wireless Networks 5. — 1999. — P. 195–209.
2. **Johansson Ö.** Simple distributed $\Delta + 1$ — coloring of graphs // Inf. Process. Letters. — 1999. — Vol. 70. — P. 229–232.
3. **Grable D.A., Panconesi A.** Fast distributed algorithms for Brooks-Vizing colorings // J. Algorithms 37. — P. 85–120.
4. **Hansen J., Kubale M., Kuszner L. Nadolski A.** Distributed Largest-first algorithm for graph coloring // Proc. of EuroPar 2004. — Lect. Notes Comput. Sci. — 2004. — Vol. 3149. — P. 527–539.
5. **Szekeres G., Wief H.S.** An inequality for the chromatic number of a graph // J. Combin. Theory. — 1964. — Vol. 4. — P. 1–3.
6. **Волошин В.И.** Свойство триангулированных графов // Исслед. операций и программирования мат.наук. — Кишинев, 1982. — С. 24–32.
7. **Маркосян С.Е., Гаспарян Г.С.** *w*-совершенные графы // Ученые записки. — Ереван.гос.универ-т, 1987. — № 3. — С. 9–15.
8. **Евстигнеев В.А.** Хордальные графы и их свойства // Проблемы систем информатики и программирования. — Новосибирск, 1999. — С. 33–64.
9. **Евстигнеев В.А.** Применение теории графов в программировании. — М.: Наука, 1985. — 352 с.
10. **Кристофиди Н.** Теория графов. Алгоритмический подход. — М.: Мир, 1978. — 432 с.
11. **Matula D.W., Bleck L.L.** Smallest-last ordering and dustering and graph coloring algorithms // J. Assoc. Comput. Math. — 1983. — Vol. 30. N 3. — P. 417–427.
12. **Евстигнеев В.А.** О некоторых свойствах локальных алгоритмов на графах // Комбинаторно-алгебраические методы в прикладной математике. — Горький: ГГУ, 1983. — С. 72–105.

Р. И. Идрисов

ВРЕМЕННАЯ РАЗВЁРТКА ВНУТРЕННЕГО ПРЕДСТАВЛЕНИЯ IR2 ЯЗЫКА SISAL 3.1*

На сегодняшний день увеличение вычислительных мощностей связано уже не с ускорением отдельного, а с добавлением дополнительных вычислителей, созданием различного рода суперкомпьютеров и кластеров; в связи с этим, распараллеливание программ становится более актуальным. Для человека, решающего конкретную вычислительную задачу, удобнее всего не вдаваться в конкретные детали распараллеливания своей задачи и иметь платформенно-независимый код. Одним из возможных решений является использование специализированных языков, которые легко распараллеливаются.

Язык Sisal [1] реализует потоковую модель вычислений и является одним из самых известных языков такого типа. Он также позиционируется как замена языка Fortran для вычислений, поскольку в отличие от других потоковых языков имеет синтаксис, более схожий с привычными языками программирования, такими как Pascal. Потоковая организация вычислений обеспечивает более естественное распараллеливание кода. Механизм однократного присваивания сильно упрощает анализ зависимостей.

Компилятор языка, разрабатываемый в Институте систем информатики им. А. П. Ершова (ИСИ СО РАН), имеет только последовательную реализацию и не имеет средств к оптимизации и выявлению параллелизма. В связи с этим задача распараллеливания оптимизации Sisal является актуальной.

Целью данной работы является формулирование задачи распараллеливания в терминах внутреннего представления языка Sisal IR2.

ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ

Для исследования параллельных свойств программ, записанных на императивных языках, используются **графы зависимостей** [2]. Графом зависимостей называется граф, построенный для некоторой программы, в котором вершины соответствуют её операторам, а дуги соединяют две вершины

* Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

в случае, если соответствующие операторы используют одну переменную. Минимальным снизу графом зависимостей называется граф, в котором вершина имеет только по одной входящей дуге для каждой из используемых переменных, и эта дуга идёт из ближайшей предшествующей по исполнению к ней вершины (так как программа последовательная, и порядок чётко определён, ближайшая вершина — единственная). Для каждого из четырёх видов зависимостей можно построить такой граф. Мы будем рассматривать императивные программы только с однократным присваиванием, поэтому нас интересует исключительно граф истинных зависимостей. Далее термин *граф зависимостей* будет обозначать минимальный снизу граф истинных зависимостей.

В языке Sisal на данный момент используется внутреннее представление IR2 [3], которое не содержит указания на последовательность кода и не зависит от архитектуры целевого компьютера; оно представляет собой иерархический граф, в котором каждая вершина означает функцию, а ребро — передачу информации. Граф реализован с помощью двух сущностей: вершина и порт. Порт отвечает за входные и выходные данные и связан с другим портом — поставщиком или получателем данных. Вершина может содержать входные и выходные порты и также подграфы, относящиеся к внутренним вычислениям, связь по данным с которыми также осуществляется посредством портов. Вершина может не содержать входных портов; такая вершина отвечает за константное значение. Сущности дуги как таковой нет, но можно считать, что дуга соединяет порты, между которыми происходит передача данных. Нашей задачей является формулировка условий задачи распараллеливания в терминах IR2 и отображение архитектурных особенностей на условия в терминах IR2.

ПОСТРОЕНИЕ РАЗВЁРТКИ

Для последовательных программ на императивных языках, представленных в виде минимального сверху графа зависимостей по данным G, временной строгой развёрткой называется вещественный функционал f(G), который возрастает вдоль дуг графа. Это означает, что если из вершины u идёт дуга в вершину v, то $f(u) < f(v)$. Для учёта реальных условий добавляются следующие векторы: h_i — вектор реализации, отвечающий за скорость выполнения операций в вершинах графа, w_{ij} — вектор задержек, отвечающий за задержки при передаче данных между вершинами и вектор s_i — вектор граничных значений, определённый для входных вершин, отвечающий

за подачу данных. Все значения ≥0, индексы i, j могут принимать значения от 1 до N, где N — число вершин графа. Временная развёртка может быть представлена набором чисел t_i, где $t_i = s_i$ для входных вершин, а для остальных $t_i >= \max(t_j + w_{ij}) + h_i$, где j принимает значения соответствующе смежным вершинам.

Временной развёрткой для внутреннего представления языка Sisal будем называть вектор t_i, где $i \in [1..N]$ и N — количество портов внутреннего представления, удовлетворяющий следующим условиям:

1. если данные передаются из порта p_i в порт p_k, то $t_i \leq t_k$,
2. если p_i — входной, а p_k — выходной порт одной вершины, то $t_i \leq t_k$.

Аналогичным образом введём вектор граничных значений s_i, который определяет значения t_i для входных портов, не связанных с другими портами, вектор задержек w_{ij} и вектор реализации h_i. В случае графа зависимости смысл векторов w_{ij} и h_i прозрачен; требуется установить их смысл для представления IR2, поскольку условия, накладываемые архитектурой, удобней формулировать в терминах выполнения конкретных операций и задержек на пересылку.

Для представления, которое содержит только простые вершины (не содержащие подграфов), построим соответствующий граф зависимостей следующим образом: каждому входному порту будет соответствовать входная вершина графа зависимостей, каждой вершине — вершина в новом графе, дуги соединяют вершины в случае, если в исходном графе соответствующие вершины были соединены через порты. Если найдена временная развёртка графа зависимости по данным, удовлетворяющая ограничительным векторам, из неё можно получить временную развёртку исходного внутреннего представления IR2 следующим образом: для входных портов компоненты вектора t примем равными значениям развёртки в вершинах графа зависимостей, для выходных портов вершины t примем равным $t_k + h_k$, где k — номер соответствующей вершины. Времена реализации этих развёрток $\max(t_i)$ будут совпадать. Таким образом, в этом случае можно сказать, что вектор реализации h_i, определённый для вершин IR2 обозначает скорость выполнения операции, расположенной в вершине, а вектор w_{ij}, определённый только для соседних портов — задержки на пересылку данных.

Для каждой вершины, содержащей подграф, можно построить граф зависимости, поскольку возможна её трансляция в последовательную программу на императивном языке, а для такой программы граф зависимостей может быть построен. Вопрос возникает для задержек wij, которые относятся к портам вершины, соединяющих внутреннюю часть с внешней. Для

таких портов будем считать внутреннюю часть пересылки фиктивной и её задержку равной 0, поскольку нет смысла осуществлять двойную пересылку данных в данной модели. Вектор реализации для составных вершин не относится напрямую к архитектурным особенностям целевого вычислителя, а скорее к особенностям подграфа, который содержится в составной вершине. Особенностью реализации представления IR2 является также то, что внутренние подграфы вершин не всегда связаны портами со своей (родительской) вершиной, хотя такая связь предполагается. Например, вершина, соответствующая циклу, содержит четыре подграфа: граф инвариантов цикла, граф тела цикла, граф постусловия и граф генерации выходного значения. Граф постусловия получает на вход кроме значений, сгенерированных в графе инвариантов, также значения, сгенерированные на предыдущей итерации и на текущей. Фактически он должен быть вычислен после двух итераций цикла (по готовности обоих наборов значений), но вторая итерация не может быть вычислена до срабатывания постусловия. Первая итерация не может быть вычислена до срабатывания предыдущей, потому что связана с ней по данным. Из этих примеров видно, что нельзя просто соединить внутренние порты составных вершин и провести анализ практически так же, как и для графа зависимостей. Для составной вершины операции выбора вычисление графа, относящегося к условию, может быть вычислено до готовности остальных операндов, аналогично для других подграфов, составляющих вершину этой операции. Следовательно, составная вершина графа представления IR2 не может быть рассмотрена как простая (в виде макро-операции) без ограничения параллелизма. Это означает, что без изменения структуры задача не сводится к вычислению развёрток графов, составляющих представление.

Алгоритм построения развёрток, используемый для графов зависимостей, может быть использован при анализе графов IR2, которые не содержат подграфов. В этом случае начальные условия для нахождения временной развёртки можно обозначить также тремя векторами s, h и w, для которых s_i определено для входных портов и означает времена поступления начальных данных, h_i определено для вершин и означает скорость выполнения операций, w_{ij} определено для соседних портов и означает пересылку данных. Для «сшивки» развёрток на входе и выходе составной вершины будем пользоваться дополнительными правилами, учитывающими особенности представления IR2.

Задача нахождения временной развёртки программы, записанной в представлении IR2, сводится к нахождению вектора t, определённого для всех портов и означающего моменты времени готовности операнда порта.

Критерием качества распараллеливания естественно считать число $\max(t_i)$, которое означает время готовности последнего операнда.

Рассмотрим граф зависимостей некоторой программы G, состоящий как минимум из двух вершин. Разобьём множество его вершин V на два непустых множества V’ и V’’ произвольным образом. Построим графы G’ и G’’ таким образом, что G’ будет включать вершины из V’ и дуги, их соединяющие, а G’’ вершины из V’’ и дуги из вершин V’’ в вершины V’’. Для каждой дуги v_i->v_j из V’ в V’’ добавим выходную вершину, соединённую с вершиной v_i, в граф G’ и входную, соединённую с вершиной v_j, в граф G’’. Аналогично для дуг из V’’ в V’ добавим соответствующие входные и выходные вершины в графы G’ и G’’. Пусть ограничения заданы для графа G векторами s, h и w описанными выше.

Утверждение. Минимальная временная развёртка графа G может быть построена из минимальных временных развёрток для графов G’ и G’’ тогда, когда дуги, соединяющие вершины множеств V’ и V’’ в исходном графе G, имеют одну направленность (из V’ в V’’ или наоборот) или отсутствуют.

Доказательство. предположим, что дуги идут из V’ в V’’. Будем строить минимальную временную развёртку для G’. Она может быть построена, если доопределить векторы ограничений s, h и w для новых дуг и вершин, отсутствующих в G. Для добавленных выходных вершин примем $h_i = 0$, для добавленных дуг w_{ij}’= w_{ij}, где индексы i, j соответствуют вершинам концов дуги v_i -> v_j из V’ в V’’. Вектор граничных значений s дополнять не требуется, поскольку дополнительных входных вершин добавлено не было. Для построения развёртки графа G’’ нам также требуется дополнить векторы ограничений. Примем $h_i = 0$ для новых вершин и $w_{ij} = 0$ для новых дуг. Для каждой дуги v_i -> v_j из V’ в V’’ в граф G’’ была добавлена входная вершина. Примем значение вектора s_k для этой вершины равным значению временной развёртки в выходной вершине графа G’, которая была добавлена для этого ребра. Векторы ограничений дополнены, и развёртка может быть найдена. По построению все вершины графа G содержатся в G’ либо в G’’, временную развёртку для вершин графа G примем равной найденным значениям соответствующих развёрток в графах G’ и G’’. Для того, чтобы этот вектор был развёрткой графа G, необходимо выполнение двух условий: $t_i = s_i$ для входных вершин и $t_i >= \max(t_j + w_{ij}) + h_i$ (1) — для всех остальных. Первое условие выполняется по построению, второе условие для вершин, которые не имели дуг, связывающих V’ и V’’, также выполняется по построению. Остаётся проверить для всех вершин, соединённых дугой v_i -> v_j где вершина v_i принадлежит к V’, а v_j — к V’’. Для вершин v_i проверки не

требуется, поскольку в сумме участвуют только инцидентные вершины, а они никак не изменились. Для v_j по построению значение вектора развёртки будет $t_j >= \max(s_k + 0) + 0$ (2), где 0 подставлены вместо добавленных w_{ij} и h_k новых вершин и рёбер, которые мы приняли равными 0 для графа G'', а s_k-значения вектора ограничений для новых вершин. Этот вектор может быть расписан через значение развёртки в графе G' как $s_k >= \max(t_l + w_{kl}) + 0$, поскольку в каждую добавленную выходную вершину графа ведёт только одна дуга $s_k >= t_l + w_{kl}$. Из минимальности развёртки следует, что $s_k = t_l + w_{kl}$, так как значение w для новых рёбер соответствовало значению для ребра из V' в V''. При подстановке в неравенство условия (2) получим условие для развёртки (1). Верно и обратное: если развёртка графа G удовлетворяет условию (1), то она порождает развёртку для графов G' и G'' если принять вектор начальных условий s_k для добавленных вершин равным $t_l + w_{kl}$; в противном случае развёртка не будет минимальной. Таким образом, если построенная развёртка G не является минимальной, значит, есть такая вершина, для которой значение t может быть уменьшено (по определению минимальная развёртка минимальна для всех вершин графа), значит развёртка какого-то из графов G' или G'' может быть уменьшена, чего не может быть, поскольку они минимальны. Получаем, что развёртка графа G, построенная таким образом, является минимальной. Для случая, когда дуги идут из V'' в V', доказательство аналогично, в случае отсутствия дуг — тривиально.

Для построения минимальной развёртки графа G через вычисление развёрток для графов G' и G'' в случае, если в исходном графе существует дуга из V' в V'' и существует дуга из V'' в V', потребуется неоднократное вычисление минимальных развёрток графов G' и G''. Аналогичным образом добавим в графы дополнительные вершины. Развёртка графа G' может быть вычислена только для вершин, которые не связаны по пути с вершиной-источником, заменяющей ребро из графа G''. Для графа G'' аналогично. Последовательным вычислением развёрток мы можем дополнять векторы граничных значений графов G' и G'' пока полная развёртка не будет найдена. Нахождение развёртки таким способом не завершится, если в графе G присутствовал контур, и часть его оказалась в G', а другая в G''. Но мы рассматриваем построение развёрток только для бесконтурных графов. Количество итераций не будет превосходить $\min(n_f, n_b) + 1$, где n_f,n_b количество прямых и обратных дуг соединяющих вершины V' с V'' в исходном графе G. Построенная таким образом развёртка будет развёрткой для графа G, способ и доказательство аналогично случаю для одного типа рёбер. На каждом шаге вычисления будут производиться не для всего графа G' или

G’’, а только для тех вершин, которые достижимы из вершины-источника, начальное условие для которой было определено на предыдущей итерации.

ЗАКЛЮЧЕНИЕ

Для программы, записанной в терминах внутреннего представления IR2, задача вычисления временной развёртки портов может быть сведена к задаче отдельного вычисления развёрток для его подграфов. Потребуется представление составных вершин представления графа в виде графа зависимости. Такая развёртка будет минимальной. Существенно то, что нам не обязательно приводить всю Sisal программу к виду графа зависимости, а можно ограничиться только определением структур для составных вершин. Решение задачи для портов в случае простых вершин, не содержащих подграфы, аналогично решению задачи для вершин графа зависимости.

Решение задачи о минимальной временной развёртке позволяет определить минимальное время исполнения программы на граф-машине или в условиях неограниченного параллелизма. Это время даёт оценку снизу на время исполнения программы; если значение получается неприемлемо большим — требуется изменение программы для того, чтобы компилятор смог выделить больше параллельных участков кода. Сравнение значения для временной развёртки до и после оптимизационных преобразований может служить критерием их эффективности для данной программы.

СПИСОК ЛИТЕРАТУРЫ

1. Касьянов В. Н., Бирюкова Ю. В., Евстигнеев В. А. Функциональный язык Sisal // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск: ИСИ СО РАН, 2001. — С. 54–67.
2. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.
3. Стасенко А. П. Внутреннее представление системы функционального программирования Sisal 3.0 — Новосибирск, 2004. — 54 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 110).

Р. И. Идрисов

МЕТОДЫ МЕЖПРОЦЕДУРНОГО АНАЛИЗА[*]

1. ВВЕДЕНИЕ

Межпроцедурный анализ относится в первую очередь к анализу потока данных, который поступает при вызове в процедуру и из неё. Анализ используется для отслеживания передачи константных значений, данных, которые содержатся в одной ячейке, областей массивов. Такой вид анализа используется для контролирования системы типов в средах разработки и при выполнении преобразований в оптимизирующем компиляторе. Можно обойтись без межпроцедурного анализа, если осуществлять подстановку тела процедуры на место вызова (inlining). Это приводит к экспоненциальному увеличению анализируемого кода, но открывает возможность использования обычных методик анализа. Подстановку нельзя реализовать в полной мере, когда граф вызовов содержит контуры, поскольку это потребует неограниченного количества памяти. При частичной подстановке за счёт удаления мёртвого кода глубина рекурсии может быть определена на стадии компиляции, но размер анализируемого кода и в этом случае увеличится экспоненциально. Межпроцедурный анализ приобретает особую ценность в распараллеливающих компиляторах. Если не анализировать код вызываемых процедур, придётся предположить, что все параметры и глобальные переменные могут измениться в результате вызова. Это существенно снизит эффективность результирующего кода, потому что, например, циклы, содержащие вызовы, не будут распараллелены никогда. Алгоритмы межпроцедурного анализа зачастую являются трудоёмкими. Требуется сохранить баланс между скоростью проведения анализа и эффективностью распараллеливания, что сейчас и демонстрируют основные распараллеливающие системы. Целью данной работы является обзор существующих методик, выбор наиболее перспективных алгоритмов и направлений для развития в межпроцедурном анализе.

[*] Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

При подготовке обзора использовались публикации по основным системам автоматического распараллеливания FIAT, Polaris, PIPS, Fida, Parafrase-2, R^N, Parascope, PTRAN.

2. ОСНОВНЫЕ МЕТОДИКИ МЕЖПРОЦЕДУРНОГО АНАЛИЗА

Межпроцедурный анализ можно разбить на четыре части: анализ совмещений (alias analysis), протягивание констант (constant propagation), анализ использования переменных и анализ контекста использования. Такое разбиение условно. Эта информация может быть вычислена для каждой процедуры в программе, что позволяет уменьшить объём компиляции, необходимой при изменении кода одной из процедур. В визуальных системах программирования полезно получать эту информацию как можно быстрей для того, чтобы давать пользователю своевременные подсказки.

2.1. Анализ совмещений

Анализ совмещений. (Alias Analysis) [1], [2] помогает предотвратить появление неверного кода в результате оптимизационных преобразований. Например, последовательность присвоений $a = 5$; $b = 3$; $c = a * b$; можно оптимизировать как $c = 15$ при условии, что a и b нигде далее не используются, но это будет неверно, если a и b хранятся в одной ячейке памяти. В таком случае, после присвоения b значения 3, переменная a тоже примет значение 3, и результат будет равен 9. Также анализ совмещений может быть использован в системах разработки программного обеспечения. При разработке больших программных проектов иногда возникает необходимость изменения типа переменной или объекта, для сохранения корректности программы и исключения нежелательных конверсий типов используют анализ совмещений. Обычно выделяют три типа совмещений:

1. **Статическое совмещение** (explicit aliasing) — возникает, когда две переменные с помощью конструкций языка обозначаются как указывающие на одну ячейку памяти (например union в языке C или equivalence в языке Fortran). Анализ таких совмещений не вызывает сложностей.
2. **Совмещение через параметры** (parameter aliasing) — возникает, когда переменная передаётся в функцию в качестве нескольких из формальных параметров или выступает в качестве параметра, но также доступна из глобального окружения.

3. **Динамическое совмещение** или совмещение указателей (pointer aliasing) — возникает вследствие неизвестных значений переменных типа “указатель”, которые могут отвечать также за одну ячейку памяти.

Рассмотрим совмещение по параметрам и динамические совмещения более подробно.

2.1.1. Совмещение через параметры

Для анализа совмещений через параметры в случае отсутствия контуров в графе вызовов программы достаточно обойти граф в прямом направлении и проанализировать каждую передачу параметра по ссылке на предмет одной и той же глобальной переменной. В этом случае мы получим множество пар переменных, которые могут быть совмещены в процессе выполнения программы. Не все эти совмещения могут возникать на практике, потому что алгоритм не учитывает условных переходов и различных контекстов вызова процедуры. В случае если в графе присутствуют контуры, можно использовать итеративный вариант алгоритма.

Для демонстрации алгоритма рассмотрим следующий пример программы:

```
Procedure p(var a,b,c,d,e: integer)
Begin
 If e=0 then exit;
 B=a+b;
 P(d,a,b,c,e-1);
End;

Begin
…
For i=0 to 9 do
 P(a[i*3], a[i*3+1], a[i*3+2], a[i*3+3],4);

End.
```

В данном примере процедура P, при входном параметре $e = 4$ вычисляет ряд частичных сумм (a, $a + b$, $a + b + c$, $a + b + c + d$). В результате работы данного участка программы каждый элемент массива будет дополнен суммой предшествующих элементов. Совмещений по параметрам в этом

случае не возникает, но если изменить вызов процедуры следующим образом:

```
Begin
…
For i=0 to 9 do
 P(a[i*3], a[i*3+1], a[i*3+1], a[i*3+2],4);

End.
```

Возникает совмещение b и c при первом вызове. Продемонстрируем действие итеративного алгоритма поиска совмещений. Граф вызовов для данной программы содержит петлю в вершине, относящейся к процедуре p, алгоритм при рассмотрении каждой процедуры строит множество совмещений, которое получается при её вызове, и протягивает эти данные для вызываемых процедур. Завершение происходит, когда на каком-то из шагов не происходит изменения множества совмещений. На первом шаге анализа процедуры множества совмещений выглядят следующим образом:

```
a->a
b->b,c
c->c,b
d->d
e->e
```

При анализе рекурсивного вызова процедуры множества меняются следующим образом:

```
a->a,b,c
b->a,b,c
c->a,b,c
d->d
e->e
```

Переменная *a* добавляется к множеству совмещаемых переменных *b* и *c*, потому что используется вторым аргументом при вызове функции. Алгоритм завершается после следующего шага и в результате получается, что первые четыре аргумента могут быть совмещены друг с другом.

Этот алгоритм не является точным, поскольку не учитывает возможности вызова процедуры из различных контекстов; реального совмещения всех аргументов при каждом вызове не происходит. Результат нужно рас-

сматривать как множество возможных совмещений переменной. Если переменные не содержатся в одном множестве совмещений, они не могут соответствовать одной ячейке памяти в ходе выполнения программы, а если содержатся — могут соответствовать, но не обязательно соответствуют. Можно увидеть, что результирующие множества никак не зависят от параметра *e*, но если задать $e = 0$ реального совмещения параметров *a* и *b* в ходе выполнения программы не будет, для выявления таких случаев анализ совмещений иногда объединяют с алгоритмом протягивания констант. Значения констант протягиваются вглубь графа вызова аналогично информации о совмещениях, что делает удобным объединение этих двух видов анализа [3].

Вернёмся к случаю $e = 4$. Если учитывать контексты вызова, можно разделить процедуру на несколько, значения совмещений для которых будут различными, а тела — одинаковыми. Такой анализ называется контекстно-чувствительным.

Можно заметить, что стандартный итеративный алгоритм имеет достаточное количество минусов.

2.1.2. Динамическое совмещение

Наибольший интерес и сложность представляет анализ динамических совмещений (совмещений указателей). Информация о совмещениях по параметру действительна на всём протяжении процедуры, а динамические совмещения могут быть различны; они могут быть вычислены для каждого узла (оператора) отдельно. Такой подход даёт более точные результаты, но имеет большие накладные расходы. Его называют узловым (flow-sensitive) анализом. Динамические совмещения могут быть вычислены для процедуры в целом. Такой алгоритм анализа менее точен, но осуществляется с гораздо меньшими затратами (flow-insensitive alias analysis). Как и в случае совмещений по параметру, алгоритмы могут быть чувствительны к пути исполнения (context-sensitive). Такие алгоритмы в русскоязычной литературе называются контекстно-чувствительными. Нечувствительные к пути исполнения алгоритмы дают большой выигрыш в скорости анализа, они исполняются за линейное время, это объясняет их большую распространённость.

Реализовать алгоритм можно при помощи alias-переменных, которые сопоставляются также всем переменным, с которыми данная переменная может быть совмещена в ходе выполнения программы. Таким образом строятся классы эквивалентности. Здесь усматривается аналогия с анализом совмещений по параметрам, только для сбора данных о возможных совме-

щениях нужно проанализировать не только вызовы процедур, но и их код. В случае присутствия операций именования/разыменования вводятся дополнительные фиктивные переменные.

```
Int *b,*c, *d, e;
 b=&c;
 e=*d;
 d=b;
```

В данном случае совмещаются **b* и *c*, **d* и **b*, для **d* и *&c* будут созданы дополнительные фиктивные переменные. Динамические совмещения можно анализировать при помощи итеративного алгоритма, аналогичного алгоритму анализа совмещений по параметру, описанного в 2.1.1. Различия заключаются в определении наличия совмещения, например:

```
Int  a[20],*p, i;
p=&a[0];
For (i=0;i<10;i++) {
  *p=i*5;
    p++;
}
```

В данном случае при выполнении программы указатель *p* будет совмещён с первыми 10 элементами массива *a*, простой анализ совмещений укажет только на первый элемент, а более сложный, но не учитывающий значения констант/границ циклов покажет на возможное совмещение указателя *p* со всеми элементами массива. Анализ получается более сложным, если пытаться точно вычислить совмещения в случаях прямого изменения указателя *p*. В случае, если к значению указателя добавляется число, которое программа получает в качестве входных данных, или случайное число, совмещения не могут быть вычислены точно на этапе статического анализа. Это не единственный случай, в котором невозможно точно вычислить множества совмещений, как и для совмещений по параметру наличие переменной во множестве указывает только на возможное совмещение в ходе выполнения программы, но не на обязательное. Существуют алгоритмы, которые создают различные множества для возможных и обязательных совмещений (переменные, которые будут обязательно совмещены в ходе выполнения программы). Такой анализ является более точным.

В системах разработки программного обеспечения используются алгоритмы, которые способны извлекать информацию не только о переменных, ссылающихся на одну ячейку памяти, но и о переменных, для которых происходит присвоение. Это производится для контроля типов. Например:

```
Int a,b,c,d;
A=5;
B=7;
C=a+b;
D=c*b;
```

При изменении типа переменной *c* на real желательно изменить тип переменной *d* для того, чтобы избежать нежелательной конверсии типов. Такой анализ не является анализом совмещений в чистом виде, но называется так же. Алгоритмы в этом случае классифицируются как отождествляющие (unification-based), для которых наличие присвоения $y = x$ в теле программы вызывает отождествление вершин y и x в графе совмещений (points-to graph), и «не отождествляющие» (subset-based) алгоритмы, для которых аналогичный случай создаёт зависимость $y \in x$ в графе совмещений. Обычно отождествляющие алгоритмы хранят информацию о совмещениях в виде множеств, alias- переменных или пар возможных совмещений, построение ориентированного графа совмещений характерно для не отождествляющих алгоритмов. Эта информация позволяет проследить цепочку получения значения другого типа. Отождествляющие алгоритмы дают более грубый результат, но исполняются за линейное время. «Не отождествляющие» имеют полиномиальную сложность $O(n^3)$ [7], но предоставляют гораздо более детальную информацию о совмещениях переменной. В системах визуального программирования предпочтительно использование «не отождествляющих» алгоритмов, поскольку по такой информации пользователю будет гораздо проще ориентироваться в динамических совмещениях, возникающих в коде программы. В системах, которые предназначены для анализа больших объёмов кода, иногда используются индексированные базы данных [2] для хранения межпроцедурной информации и быстрого доступа оптимизирующих алгоритмов.

2.2. Распространение констант

В некоторых случаях значения или множество возможных значений переменных может быть определено на стадии компиляции программы. Строго говоря, множество значений переменной всегда ограничено её типом и может быть определено на стадии компиляции, но в данном изложении будем считать, что множество значений переменной в таких случаях не определено. Рассмотрим следующий пример:

```
Procedure P(var a,b:integer);
Begin
 If a<50 then exit;
 b=a+b;
End;
…
Begin
 …
i=5;
P(i,j);
…
For i=1 to 20 do
 P(i,j);

End.
```

В этом примере процедура всегда вызывается с параметром $a < 50$, и, следовательно, может быть полностью удалена, потому что суммирование $b = a + b$ не выполняется никогда. В случаях, когда значение переменной может быть определено на стадии компиляции, используется следующая методика:

Распространение констант (constant propagation) — распространение информации о возможных значениях переменной внутрь процедуры, осуществляемое на стадии компиляции. В случае, когда значение единственное, осуществляется замена переменной её значением в теле процедуры [4, 5].

Информация о значениях используется при анализе индуктивных переменных, границ и шагов циклов. Через границы циклов могут быть определены используемые области массивов [6]. В работах PIPS информация о возможных значениях переменной называется начальными условиями (preconditions) и вычисляется для каждого из анализируемых контекстов. Начальное условие не обязательно представляет информацию о конкретном значении переменной; это может быть множество возможных значений переменной. Эта информация может быть очень полезной при анализе возможности распараллеливания цикла. Даже то, что переменная принимает значения строго больше ноля, может сказаться на возможности параллельного исполнения итераций цикла. Также в статьях о системе PIPS вводится такое понятие, как преобразователи (transformers), которое отражает характер изменения переменной в результате выполнения операции. Преобразо-

ватель — функция, определённая для каждого изменяемого параметра функции, определённой в языке программирования, преобразующая множество значений параметра до выполнения функции во множество возможных значений после её выполнения. Начальные условия для следующей из последовательно исполняемых команд получаются в результате действия преобразователя предыдущей команды на начальные условия для неё. Например, если задано начальное условие i > 0 для операции $i = i + 1$, тогда начальные условия для следующей операции будут i > 1. Начальные условия распространяются в прямом направлении, а преобразователи — в обратном.

Простейший алгоритм анализа начальных условий и преобразователей, при отсутствии в графе вызовов циклов, можно осуществить в два прохода. На первом (обратном) проходе можно получить все преобразователи, затем на втором (прямом) проходе получить все начальные условия. Конечно, здесь не имеется в виду, что преобразователи вычисляются точно, это не всегда возможно, потому что в точную функцию преобразователя войдут параметры, которые невозможно вычислить на стадии компиляции, и сама функция будет являться срезом процедуры относительно параметра [7].

Для распространения констант можно использовать итеративный алгоритм, аналогичный 2.1.2, основные отличия алгоритмов, используемых в оптимизирующих компиляторах, заключаются в представлении множества возможных значений переменной.

1) значения переменных представлены с помощью единственного значения, если такое может быть вычислено,
2) значения переменных представлены в виде интервала, например [0, 9] — означает, что переменная может принимать значения внутри этого интервала,
3) значения переменных представлены в виде множественных интервалов,
4) значения переменных представлены в форме kx + b с заданным шагом k, смещением b и диапазоном изменения x в виде интервала;

Также должна быть определена операция объединения двух множеств значений переменной и преобразователи для функций языка.

Иногда распространение констант объединяют с анализом совмещений, называя протягиванием межпроцедурной информации, а остальную часть межпроцедурного анализа — анализом использования данных. Особую важность распространение констант имеет вследствие того, что границы

массивов отсчитываются по границам изменения индексных переменных, которые могут быть переданы внутрь процедуры в качестве параметра.

2.3. Анализ использования переменных

При анализе процедуры нам будут интересны четыре множества переменных:

1) READ — множество переменных, используемых для чтения в теле подпрограммы,
2) WRITE — множество переменных, используемых для записи в теле подпрограммы,
3) IN –множество используемых для чтения внешних переменных и параметров процедуры,
4) OUT — множество переменных, вычисляемых и записываемых в процедуре, используемых последующем коде.

Первые три множества могут быть получены при анализе тела подпрограммы и вызываемых ею подпрограмм, а четвёртое множество — только путём анализа всего кода, который исполняется после процедурного вызова. Такой анализ не производится, если не учитывается возможность вызова процедуры из различных контекстов и влияние контекста на тело процедуры вообще. Множество OUT отражает характеристики не самой подпрограммы, а последующего кода, и может быть использовано для межпроцедурных оптимизаций. Например, может оказаться, что некоторые или все внутрипроцедурные вычисления могут быть удалены как мёртвый код.

Множества READ, WRITE и IN можно получить при помощи итеративного алгоритма в один проход:

1. Изначально все множества принимаются пустыми.
2. Для каждого узла:
   - если присутствует чтение из некоторой переменной, и она не содержится во множестве WRITE — её следует добавить во множество READ;
   - если добавленная переменная не является локальной — она добавляется в IN;
   - если присутствует запись в некоторую переменную — её следует добавить во множество WRITE;
   - если переменная, добавленная во множество WRITE, не является локальной — она добавляется так же в OUT.

3. Для каждого вызова подпрограммы потребуется предварительное вычисление его множеств IN и OUT (хотя бы приблизительное), далее вызов рассматривается как обычный узел, использующий переменные IN и вычисляющий (записывающий) переменные OUT.

Здесь под локальными переменными понимаются переменные, область видимости которых ограничивается данной процедурой. В результате мы получаем первые три множества и верхнюю оценку множества OUT, для точного вычисления которого требуется проанализировать остаток кода программы на предмет использования (чтения) переменных этого множества. Запись без чтения автоматически убирает переменную из множества OUT, поскольку в этом случае результат вычисления не используется, с оговоркой, что запись производится для любого из контекстов вызова процедуры (для любого пути исполнения в графе вызовов программы).

Если при анализе переменных рассматривать каждый массив как одно целое (запись в элемент массива считать записью массива, а чтение переменной массива считать чтением массива) — результат получается слишком грубым. Для более точного анализа следует рассматривать компоненты массива отдельно, что и производится при анализе итераций циклов, использующих массивы, но для глобального анализа этот способ имеет слишком много накладных расходов. Компромиссом является рассмотрение отдельных областей массива в качестве атомарных единиц. Существует множество способов описания областей, и всегда можно выбрать тот, который наиболее подходит для конкретной системы по точности/скорости работы и требуемым объёмам памяти. Рассмотрим основные способы описания областей массивов. Их можно разделить на две группы:

1) точные — дают такой же результат, что и анализ каждой компоненты массива как отдельной переменной;
2) неточные — дают приближённое описание областей массивов, которое требует меньше памяти и вычислительного времени, но даёт огрублённый результат.

2.3.1. Неточные алгоритмы описания областей массивов

Кроме рассмотренных ниже алгоритмов анализа, разработчики Fida [8] включают в состав неточных алгоритмов “Пессимистический (Pessimistic) алгоритм”, который заключается в том, что все массивы предполагаются изменяемыми в процессе работы процедуры (межпроцедурный анализ не осуществляется), и “Классический (Classic)” — анализ массивов как скаляров. Это делается для сравнения их на тестах другими алгоритмами.

2.3.1.1. Регулярные секции (Regular Sections)

Этот алгоритм впервые предложили Каллаган и Кеннеди [9] (Callahan, Kennedy). Области массивов задаются через значения индексных переменных размерностей массива. Регулярные секции делятся на два вида: ограниченные регулярные секции (restricted regular sections) и описания с помощью триплетов (bounded regular sections). В методе ограниченных регулярных секций каждой из размерностей массива сопоставляется одно из следующих выражений:

1) $\perp$ — если индексная переменная может принимать любое значение;
2) α — если переменная принимает только одно, константное значение;
3) $\alpha \pm j_k$ — если индексная переменная связана с другой индексной переменной этого же массива константным смещением.

Таким образом, этот алгоритм может описывать строки, диагонали и ещё некоторые виды регионов в массиве, но несложно придумать такую область, описание которой даст весь массив целиком, потому что не найдётся другого возможного описания. При объединении и пересечении областей требуется процесс нормализации (приведения к зависимости от одной индексной переменной в случае, если есть записи типа 3). Сложность алгоритма объединения областей — $O(d^2)$, где d — размерность массива.

При описании с помощью триплетов (bounded regular sections) каждой размерности массива сопоставляется тройка чисел (l:u:s), где l — нижняя граница значений индексной переменной, u — верхняя граница, а s — шаг изменения значения индексной переменной. Также возможны значения (α.α.α) — если индексная переменная принимает константное значение, ($\perp$.$\perp$.$\perp$) — если значение индексной переменной неизвестно. Этот алгоритм не требует нормализации при объединении областей, его сложность — O(d).

2.3.1.2. Дескрипторы доступа к данным (Data Access Descriptors)

Впервые предложен Валасундарам и Кеннеди [10] (Balasundaram, Kennedy). В алгоритме области массива описываются простыми секциями (simple sections), которые имеют ортогональные и диагональные границы. Ортогональные границы накладывают условие на максимальное и минимальное значение индексной переменной размерности, диагонали накладывают ограничение на пару индексных переменных различных размерностей. Границы хранятся в следующем виде:

1) $\alpha \le x_i \le \beta, i \in [1,n]F$ — ортогональная граница,
2) $\alpha \le x_i - x_j \le \beta, \forall i, j \in [1,n], i \ne jF$ — диагональ,
3) $\alpha \le x_i + x_j \le \beta, \forall i, j \in [1,n], i \ne jF$ — обратная диагональ.

Время построения минимальной оболочки для набора циклов не может быть выражено через их количество и размерность массива. Скорость построения объединения O(d) согласно [11], где d — количество уравнений в описании области массива.

2.3.1.3. Регионы (Regions)

Алгоритм заключается в описании областей массива в виде минимальной выпуклой оболочки (convex hull) [12], которая ограничивается линейными неравенствами. Неравенства делятся на набор индексных выражений (region) и контекст исполнения (execution context). Контекст исполнения отражает ограничения на управляющие параметры цикла. Здесь, в отличие от предыдущего метода, линейные неравенства не ограничиваются только диагональными и ортогональными границами. Это позволяет описать более сложные области, но увеличивает время построения оболочки. Скорость объединения также O(d) согласно [11].

2.3.2. Точные алгоритмы описания областей массивов

2.3.2.1. Образы (Atom Images)

Этот алгоритм впервые предложен Ли и Ю [3]. Идея — описать области доступа к массиву с помощью неравенств на каждую из индексных переменных. Предлагаются две структуры для хранения этих данных:

- **Атом** хранит информацию о ссылке на массив, получаемую из самой ссылки. Это двумерный массив, в котором строки соответствуют индексным размерностям массива, а столбцы строки — коэффициентам при индексных переменных в выражении, используемом для доступа к массиву. Нулевой столбец содержит константу — инвариант цикла. Также с каждым рядом ассоциируется флаг, который отвечает за линейность размерности.
- **Атомное изображение** аналогично **атому**, но содержит также информацию о пределах изменения индексных переменных. Эта информация дописывается в конец массива: после k рядов атома (где k — количество объемлющих циклов), идёт ряд k + 1, в котором находится линейное выражение, отвечающее за верхнюю границу изменения переменной 1,

аналогично для k + i. Все циклы предполагаются нормализованными, начинающимися с 1.

Этим методом точно описываются области доступа к массиву, потому что границы получаются напрямую из границ изменения индексных переменных объемлющих циклов. Главный недостаток такого алгоритма в том, что невозможно получить объединение двух областей. В этом заключается принципиальное отличие от метода **2.3.1.1 Регулярных секций**.

2.3.2.2. Линеаризация (Linearization)

Этот подход предложен Бурке и Сайтроном [13] (Burke, Cytron). В методе память рассматривается как одномерный массив, и все ссылки на массивы преобразуются в ссылки на память. При слиянии двух областей можно огрублять результат, предполагая, что результирующая область полностью занимает всё пространство между дальними границами областей массивов, но тогда метод становится неточным. Автоматически решаются некоторые проблемы анализа совмещений (Aliasing), но серьёзный недостаток этого метода в том, что требуется хранить большое количество неравенств для каждого из массивов, вследствие чего скорость проверки на зависимость понижается. При построении объединения участки одной области просто дописываются к участкам другой, а при построении пересечения нужно проанализировать $m_1 * m_2$ комбинаций, где m — количество участков, используемое для описания области.

2.3.2.3. Межпроцедурный Омега-тест (Interprocedural Omega-test)

Рассмотрим следующий пример:

```
For i=1 to 100 do
a[i,i]=a[50,i];
```

для того, чтобы показать, что области, используемые в качестве левого и правого значения присвоения пересекаются; достаточно найти целочисленное решение системы уравнений

```
i=i
50=i
```

В данном примере сразу же видно, что система имеет решение при $i = 50$, что лежит в диапазоне изменения индексной переменной. Проблема заключается в том, что целочисленное решение произвольной системы уравнений является np-полной задачей.

Метод, предложенный Тангом (Tang) заключается в том, что все границы и шаги циклов записываются в виде системы уравнений и неравенств. Для построения пересечения областей используется целочисленный метод,

являющийся расширением метода исключения переменных Фурье—Моцкина [14]. Этот алгоритм имеет экспоненциальную сложность в худшем случае, но тесты показывают, что его можно использовать в реальных компиляторах и для большинства задач время его работы полиноминально [15].

2.3.3. Комбинированные методы

В реальных компиляторах возможно использование методов, в которых комбинируются точные и не точные алгоритмы описания областей массивов. Наиболее известным комбинированным методом является FIDA (Full Interprocedural Data Dependence Analysis) Майкла Хинда (Hind [8]) из IBM. В этом методе использованы алгоритмы линеаризации и образов. Для определения зависимости производится частичная подстановка процедурного кода на место вызова (результирующий код содержит вызовы без подстановки, подстановка производится только для анализа). Данные, которые при этом не являются необходимыми (для анализа зависимости), отбрасываются (вычисляется срез [7]). Линеаризация используется в тех случаях, когда размерность массива изменяется при вызове процедуры, или используются смещения начала массива. За счёт использования подстановки алгоритм позволяет использовать стандартные методы нахождения зависимости (во многих случаях).

2.4. Анализ контекста использования процедуры

Анализ контекста использования направлен на уточнение оптимизаций путём анализа различных вариантов использования процедуры. Информация о совмещениях, диапазоны изменения переменных и, как следствие, набор возможных оптимизаций зависят от контекста вызова процедуры. Рассмотрим следующий пример:

```
function a(a,b,c)
begin
 if (c=1) a=sin(b);
 else b=asin(a);
end
```

В этом примере подпрограмма может изменять параметр *b* или параметр *a* в зависимости от значения параметра *c*. Контекст вызова процедуры определяется набором параметров совмещений (alias) и информацией о

возможных значениях переменных, которая генерируется в процессе протягивания констант. В данном примере оптимизирующий компилятор может принять решение о замене этой процедуры двумя следующими:

```
function a_1(a,b)
begin
 a=sin(b);
end

function a_2(a,b)
begin
 b=asin(a);
end
```

Будет произведена замена вызовов процедуры *a* на вызовы процедур *a_1* и *a_2* в зависимости от значения параметра *c*. Оговоримся, что такое решение принимается только в результате анализа контекста использования процедуры *a*, потому что возможны такие варианты программы, при которых такая оптимизация не будет возможной. Например:

```
Случай 1:
....
a(a,b, rand(0, 5))

Случай 2:
a(a,b,1);
a(c,d,1);
```

В первом случае явная разделимость процедуры *a* на две разные процедуры не даёт ничего, а во втором — просто не требуется (при условии, что это все вызовы *a* в программе), потому что используется только при c = 1, что даёт возможность найти и удалить «мёртвый код» ветви c <> 1. Замена одной процедуры несколькими называется клонированием (procedure cloning). Предельный случай клонирования процедур (когда процедура клонируется столько раз, сколько вызывается) является аналогом прямой подстановки (inlining). Клонирование процедур уменьшает огрубление информации о поведении процедуры в конкретном контексте. Решение о клонировании принимается на основе анализа контекстов возможного использования и тела процедуры. Разработчики SUIF утверждают, что таким обра-

зом они достигают точности прямой подстановки без излишнего увеличения размера анализируемого кода. В случае, если выполняется частичная подстановка, решение о подстановке принимается на основе анализа графа вызовов. Подстановка применяется к висячим процедурам.

Клонирование процедур и частичную подстановку относят к межпроцедурным оптимизациям [6], также медпроцедурной оптимизацией считается перенос кода за границы процедуры, который является промежуточным вариантом подстановки.

3. ЗАКЛЮЧЕНИЕ

Рассмотренные алгоритмы позволяют сказать, что для каждого из видов анализа не существует какого-либо универсального и точного решения поставленной задачи. Алгоритмы могут быть подобраны по производительности и точности. Область межпроцедурного анализа является развивающейся, различные источники используют разную терминологию. В целом отмечаются тенденции развития точных алгоритмов анализа, основанных на представлении программы в виде графа и тенденции развития быстрых алгоритмов межпроцедурного анализа совмещений для систем визуального программирования. Межпроцедурный анализ на сегодняшний день является обязательной частью современного оптимизирующего компилятора.

СПИСОК ЛИТЕРАТУРЫ

1. Steensgaard B. Points-to analysis in almost linear time // Proc. of POPL'96. — St. Petersburg Beach, Florida, January 1996. — P. 32–41.
2. Heintze N., Tardieu O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second // Proc. of 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation, Snowbird, Utah, USA, June 20-22, 2001. — ACM Press, 2001 — P. 254–263.
3. Li Z., Yew P.-Ch. Efficient Interprocedural Analysis for Program Parallelization and Restructuring // Proc. of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems, New Haven, Connecticut, July 19-21, 1988. — SIGPLAN Notices. — 1988. — Vol. 23(9). — P. 85–99
4. Schouten D.A. An Overview of interprocedural analysis technologies for high performance parallelizing compilers — Illinois, 1990 — 62 p. — (Tech. Rep. / Center for Supercomputing Research and Development. Univ. of Illinois; N 1005).
5. Евстигнеев В.А., Серебряков В.А. Методы межпроцедурного анализа (обзор) // Программирование. — 1992 — № 3. — С. 4–15.

6. Keryell R. et al. PIPS — A Workbench for Interprocedural Program Analyses and Parallelization / R. Keryell, C.Ancourt, F.Coelho, B.Creusillet, F.Irigoin, P.Jouvelot — Paris, 1996 — 24 p. — (Tech. Rep. / Centre de Recherche en Informatique. Ecole Nationale Sup'erieure des Mines de Paris)
7. Касьянов В.Н., Мирзуитова И.Л. SLICING: Срезы программ и их использование — Новосибирск, 2002 — 116 с.
8. Hind M. et al. An Empirical Study of Precise Interprocedural Array Analysis / M. Hind, M. Burke, P. Carini, S. Midkiff // Scientific Programming — 1994 — Vol. 3, N 3 — P. 255–271
9. Callahan D., Kennedy K. Analysis of Interprocedural Side Effects in a Parallel Programming Environment // J. of Parallel and Distributed Computing. — 1988. — Vol. 5. — P. 517–550.
10. Balasundaram V., Kennedy K. A technique for summarizing data access and its use in parallelism enhancing transformations // Proc. of the SIGPLAN '89 Conf. on Program Language Design and Implementation. — Portland, Orgen. June 1989. —Vol. 24 (7). — P. 41–53.
11. Антонов А.С. Современные методы межпроцедурного анализа программ // Программирование. — 1998. — № 5. — С. 3–14.
12. Triolet R., Irigoin F., Feautrier P. Direct Parallelization of Call Statements // Proc. of the SIGPLAN Symp. on Compiler Construction. — SIGPLAN Notices. — 1986. — Vol. 21 (7). — P. 176–185.
13. Burke M., Cytron R. Interprocedural dependence analysis and parallelization // Proc. of the SIGPLAN Symp. on Compiler Construction — SIGPLAN Notices. — 1986. — Vol. 26 (6). — P. 145–156.
14. Pugh W. The Omega Test: a fast and practical integer programming algorithm for dependence analysis — Univ. of Maryland, 1991 — 10 p. — (ACM 0-89791-459-7/91/0004).
15. Tang P. Exact Side Effects for Interprocedural Dependence Analysis // Proc. of the ACM Internat. Conf. on Supercomputing, Tokyo, Japan, July 1993 — P. 137–146.

В.Н. Касьянов, А.П. Стасенко

ЯЗЫК ПРОГРАММИРОВАНИЯ SISAL 3.2[1]

ВВЕДЕНИЕ

Используя традиционные языки и методы, очень трудно разработать высококачественное, переносимое программное обеспечение для параллельных компьютеров. В частности, параллельное программное обеспечение не может быть разработано с малыми затратами на последовательных компьютерах и потом перенесено на параллельные вычислительные системы без существенного переписывания и отладки. Поэтому высококачественное параллельное программное обеспечение может разрабатываться только небольшим кругом специалистов, имеющих прямой доступ к дорогостоящему оборудованию. Однако, используя языки программирования с неявным параллелизмом, такие как функциональный язык Sisal (аббревиатура с английского выражения Streams and Iterations in a Single Assignment Language) [1], можно преодолеть этот барьер и предоставить широкому кругу специалистов, которые не имеют доступа к параллельным вычислительным системам, но могут многое сделать в своих прикладных областях исследований, возможность быстрой разработки переносимых параллельных алгоритмов на своем рабочем месте.

Функциональная семантика языков программирования с неявным параллелизмом гарантирует детерминированные результаты для параллельной и последовательной реализаций — то, что невозможно гарантировать для традиционных языков, подобных языку Фортран. Более того, неявный параллелизм языка снимает необходимость переписывания исходного кода при переносе его с одного компьютера на другой. Гарантировано, что программа с неявным параллелизмом, правильно исполняющаяся на персональном компьютере, будет также давать правильные результаты на высокоскоростном параллельном или распределенном вычислителе.

Статья содержит описание текущей версии входного языка Sisal 3.2 системы функционального программирования SFP [2], работа над которой ведется в Лаборатории конструирования и оптимизации программ Института систем информатики СО РАН имени А.П. Ершова.

[1] Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

Цель проекта — предоставить прикладному программисту на его рабочем месте удобную среду для разработки функциональных программ, предназначенных для последующего исполнения на параллельных вычислительных системах, доступных через телекоммуникационные сети. В рамках этой среды программист получает возможность, с одной стороны, создавать и отлаживать программу без учета целевой параллельной архитектуры, а с другой — производить настройку отлаженной программы на ту или другую целевую параллельную архитектуру для достижения высокой эффективности исполнения разработанной программы на супервычислителе.

Язык программирования Sisal является одним из самых известных потоковых языков промышленного уровня и позиционируется как замена языка Фортран для научных применений [3]. Язык Sisal является результатом сотрудничества Ливерморской национальной лаборатории имени Лоренца, университета штата Колорадо, Манчестерского университета и корпорации DEC. Последняя спецификация языка Sisal версии 2.0 [4] датируется 1991 г. В 1995 г. появилось пользовательское описание языка Sisal 90 [5, 6], не содержащее точных спецификаций языка.

Язык Sisal имеет следующие особенности, облегчающие переход с популярных императивных языков программирования: приближенный к языку Паскаль [7] синтаксис, развитую систему типов и явно выделенные циклические выражения. Язык Sisal имеет следующие основополагающие качества: математическая правильность функций (отсутствие побочных эффектов), прозрачность ссылок имен, задающих значения, а не ячейки памяти, и однократность присваивания.

В 2001 г. в Институте систем информатики (ИСИ) имени А. П. Ершова СО РАН была разработана концепция языка Sisal 3.0 [8], развивающая язык Sisal 90, которая предполагала поддержку расширенных межмодульных взаимодействий, мультиязыкового программирования, а также возможностей предварительной обработки (preprocessing) и аннотирования для упрощения оптимизирующих преобразований.

В языке Sisal 3.1 [9] были специфицированы средства расширенных межмодульных взаимодействий и предварительной обработки программ. Вопросы, связанные с описанием мультиязыкового программирования и аннотирования программ языка Sisal 3.0, были оставлены для внедрения в последующих версиях языка. В язык были добавлены новые средства: пользовательские типы, переопределение операций и перегрузка функций. Базовая часть языка Sisal 90 была переработана для повышения её пригодности к нисходящему синтаксическому разбору и приближения её сходства

с языком Си. Для языка Sisal 3.1 был реализован компилятор переднего плана во внутреннее представление IR1 [10].

В языке Sisal 3.2, рассматриваемом в данной статье, специфицируется часть аннотаций (прагм), а идея поддержки мультиязыкового программирования выразилась во введении инородных типов и спецификации интерфейсов инородных модулей. Также в язык были добавлены возможности задания многомерных массивов, пользовательских типов с параметрами и обобщенных процедур. Также на язык в значительной мере повлияли идеи языка Sisal 2.0, которые не были использованы в языке Sisal 90.

Статья имеет следующую структуру. В разделе 1 рассматривается общая структура программы языка Sisal 3.2. В разделах 2 и 3 более подробно рассматриваются структура интерфейса модуля и структура модуля соответственно. Раздел 4 посвящен описанию типов и их операций, а в разделе 5 находится описание выражений. В разделе 6 описывается интерфейс взаимодействия с другими языками. В разделе 7 приводится строгое описание синтаксической и лексической структуры языка, а в разделе 8 находится писание формата Fibre для внешних значений, которые получает и порождает программа на языке Sisal. В разделе 9 обзорно приводится структура единицы компиляции на языке Си++, в которую предполагается транслировать модуль на языке Sisal.

1. СТРУКТУРА ПРОГРАММЫ ЯЗЫКА SISAL 3.2[2]

Программой (program) на языке Sisal называется совокупность файлов, каждый из которых является модулем (module) или интерфейсом модуля (module interface). Интерфейс модуля соответствует одному модулю программы, каждый из которых может иметь не более одного интерфейса.

Модуль на языке Sisal содержит определения и объявления процедур, типов (type) и определения контрактов (contract). Под процедурой понимается функция (function) или операция (operation). Модуль может быть задан:

- файлом с расширением «.s», содержащим текст на языке Sisal;
- файлом с расширением «.ir1»[3], содержащим объекты внутреннего представления IR1 (internal representation one);

[2] Далее, если не указано обратного, подразумевается версия языка Sisal.

[3] Компилятор переднего плана (front-end) Sisal порождает файл с расширением «.ir1» из файла с расширением «.s».

– файлом с расширением «.ir1.cpp»[4], который содержит исходный код на языке Си++ [11], удовлетворяющий требованиям раздела 9;
– бинарным объектным файлом с расширением «.obj»[5], сгенерированным компиляторами Си[6] или Фортран.

Модуль в формате «.ir1» описывает модуль на языке Sisal без потерь и позволяет избежать повторного анализа текста на языке Sisal. Модуль в формате «.ir1.cpp» теряет потоковую структуру программы, но полностью сохраняет возможности межмодульного взаимодействия с другими модулями. Модуль в формате «.obj» может поддерживать ограниченные возможности по взаимодействию с другими модулями, описанные в разделе 6.

Интерфейс модуля содержит видимые извне объявления процедур, определяемых в модуле, которому этот интерфейс соответствует. Также интерфейс модуля может содержать видимые извне объявления и определения типов и определения контрактов. Интерфейс модуля может быть задан:
– файлом с расширением «.s», содержащим текст на языке Sisal;
– файлом с расширением «.ir1».

Программа начитает исполняться с вызова функции main языка Си. Если функция main находится в модуле, транслируемом из представления IR1, то к нему добавляется функция «main» языка Си, которая преобразует входной поток символов формата Fibre, описываемого в разд. 8, во входные параметры функции main представления IR1. Результаты функции «main» выводятся в выходной поток символов в формате Fibre.

2. СТРУКТУРА ИНТЕРФЕЙСА МОДУЛЯ

Интерфейс модуля на языке Sisal начинается с ключевого слова[7] *interface*[8], за которым следует его имя, задаваемое идентификатором[9]. Это имя совпадает с именем модуля, которому данный интерфейс сопоставлен. В программе не может быть модулей с одинаковыми именами. Имена модулей, заданных файлами «.ir1.cpp» и «.obj», определяются на уровне проекта программы.

[4] Компилятор SFP порождает файл с расширением «.ir1.cpp» из файла с расширением «.ir1».

[5] В операционной системе Linux объектный файл имеет расширение «.o».

[6] Под языком Си всюду понимается подмножество языка Си++, соответствующее языку Си.

[7] Список ключевых слов находится в разделе 7.3.

[8] В тексте документа ключевые слова выделяются курсивом.

[9] Определение идентификатора находится в разделе 7.3.

Для модулей, заданных объектным файлом «.obj», после имени его интерфейса может следовать ключевое слово *in*, за которым находится имя языка единицы компиляции (compilation unit) данного объектного файла. Указание языка накладывает специфику на правила задания интерфейса модуля, описанную в разделе 6. Если язык не указан, то считается, что данный объектный файл был скомпилирован из файла с расширением «.ir1.cpp», и интерфейс модуля задаётся так, как описано в данном разделе.

Интерфейс модуля может содержать объявления процедур, объявления и определения типов, определения контрактов, а также конструкции импорта определений и объявлений типов и определений контрактов из других интерфейсов модулей[10]. Содержимое интерфейса модуля может разделяться точкой с запятой. Все объявления и определения влияют только на последующий текст интерфейса модуля. Интерфейс модуля завершается ключевыми словами «*end interface*».

Например, далее приводится пример интерфейса модуля «math», содержащего объявления функций для вычисления факториала и числа Фибоначчи (текст после двойной косой черты является комментарием до конца строки):

```
interface math
  function fact (n:integer returns integer) // факториал числа n
  function fib (n:integer returns integer) // число Фибоначчи с номером n
end interface
```

2.1. Объявления и определения типов

Определить можно переименованный и пользовательский типы. Определение переименованного типа выглядит как «*type* имя типа[11] = базовый тип», а определение пользовательского типа выглядит как «*type* имя типа := базовый тип». Язык Sisal поддерживает определение (как пользовательского типа) рекурсивных объединений (*union*); таким образом, имя определяемого объединения может участвовать в типах его тегов. Однако хотя бы один тип тега не должен зависеть от имени определяемого объединения во избежание бесконечных типов.

[10] В интерфейс модуля **A** импортируются только типы, так как функции, операции и редукции интерфейса модуля **B** непосредственно не могут быть использованы в интерфейсе модуля **A**. Также, при импортировании интерфейса модуля **A** было бы ненаглядно с помощью необъявленного имени интерфейса модуля **B** получать доступ к его содержимому. С другой стороны, также ненаглядно рассматривать объекты модуля **B** в качестве объектов модуля **A**.

[11] Подчеркиванием обозначается имя метапонятия.

```
// запись двух вещественных полей
type complex_record = record [ real, imag: real ]
// тип комплексного числа
type complex := record [ real, imag: real ]
// список целых чисел
type listi := union [ empty; item: record [value: integer; next: listi] ]
```

Объявить можно пользовательский тип с параметрами. Объявление пользовательского типа с параметрами[12] выглядит как «*type* имя типа [список имён параметров[13]] := базовый тип», где в базовом типе должны быть использованы все указанные имена параметров[14]. Определение пользовательского типа с параметрами выглядит как «имя типа [список типов параметров]».

```
// список произвольных элементов
type list[T] := union [ empty; item: record [value: T; next: list] ]
```

Для встроенных, составных и переименованных типов используется структурная эквивалентность базовых типов[15], для пользовательских типов используется эквивалентность по имени типа и имени модуля, в котором он был определён или объявлен, а для определённых пользовательских типов с параметрами дополнительно требуется эквивалентность типов соответствующих параметров. Пользовательские типы не эквивалентны встроенным, составным и переименованным типам. Пользовательские типы с параметрами не эквивалентны пользовательским типам без параметров.

```
// тип complex_record2 эквивалентен типу complex_record
type complex_record2 = record [ real2: real; imag2: real ]
// тип complex_record3 не эквивалентен типу complex_record
type complex_record3 = record [ real3: integer; imag3: integer ]
// типы someint из модулей A и B не эквивалентны
interface A type someint := integer end interface
interface B type someint := integer end interface
// тип listi не эквивалентен типу list[integer]
```

[12] Пользовательский тип с параметрами служит заменой понятия множества типов из предыдущих версий языка Sisal. Множество типов при определении фиксирует входящие в него типы, что не позволяет использовать обобщенные алгоритмы (функции с формальными параметрами, являющимися множествами типов) для других типов. Также пользовательский тип с параметров позволяет реализовывать пользовательские составные типы, что невозможно с помощью множеств типов. Функциональность рекурсивных множеств типов языка Sisal 90 восполняется рекурсивными объединениями.

[13] Имена параметров должны быть различны. По умолчанию, элементы списка разделяются запятой.

[14] Тем самым базовый тип обязан быть составным типом.

[15] Структурная эквивалентность не учитывает имена полей структур и тегов объединений. Структурная эквивалентность учитывает порядок следования типов тегов в объединениях. Инородные типы эквивалентны, если совпадает их строковое представление.

Имя типа объявляемого и определяемого не должно совпадать с именем простого встроенного типа[16]. Внутри одного модуля (включая его интерфейс) допускается только эквивалентное переопределение и переобъявление типов. Допускается эквивалентное переопределение переименованных, пользовательских типов и эквивалентное переобъявление пользовательских типов с параметрами. Под переопределением (переобъявлением) типа подразумевается определение типа, для которого существует предыдущее определение (объявление) типа в том же модуле (или его интерфейсе) с тем же именем. Под эквивалентным переопределением переименованного типа понимается определение переименованного типа с базовым типом, эквивалентным базовому типу предыдущего определения типа. Под эквивалентным переопределением пользовательского типа понимается определение пользовательского типа с базовым типом, эквивалентным базовому типу предыдущего определения типа. Под эквивалентным переобъявлением пользовательского типа с параметрами понимается объявление пользовательского типа с параметрами с числом параметров, равным числу параметров предыдущего объявления типа, и базовым типом, эквивалентным базовому типу предыдущего объявления типа с точностью до имён соответствующих параметров.

```
// эквивалентное переопределение переименованного типа
type complex_record = record [ real, imag: real ]
type complex_record = record [ real2: real; imag2: real ]
// эквивалентное переопределение пользовательского типа
type complex := record [ real, imag: real ]
type complex := record [ real2: real; imag2: real ]
// эквивалентное переопределение пользовательского типа с параметрами
type somerec[T1, T2, T3] := record [ a:T1; b:T2; c:T3 ]
type somerec[T3, T2, T1] := record [ a:T3; b:T2; c:T1 ]
type somerec[B, A, C] := record [ a:B; b:A; c:C ]
```

2.2. Объявления функций

Объявление функции выглядит как «*function* имя функции (список формальных параметров *returns* список типов возвращаемых значений)», где необязательные имена формальных параметров игнорируются компилятором[17]. Формой функции называется список типов её формальных параметров. Формой возвращаемых значений функции называется список типов её возвращаемых значений. Две формы функции равны, если число их типов

[16] Простые встроенные типы описываются в разделе 4.1.

[17] На самом деле, в случае несоответствия имени формального параметра в объявлении и определении функции (операции и редукции), выдаётся предупреждение.

совпадает, а соответствующие типы эквивалентны между собой. Функциями, неоднозначными по форме возвращаемых значений, называются функции, у которых совпадают имя, формы формальных параметров и различные формы возвращаемых значений.

```
// функция решения квадратного уравнения A*x^2+B*x+C
function solve_sqr_eq(A: real, B: real, C: real returns real, real)
// функции, возвращающие модуль целого и вещественного значений
function abs(integer returns integer)
function abs(real returns real)
// функции, возвращающие наибольшее из целых и вещественных значений
function max(integer, integer returns integer)
function max(real, real returns real)
function max(integer, integer, integer returns integer)
function max(real, real, real returns real)
// функции, возвращающие максимальное целое и вещественное значения
function max(returns integer)
function max(returns real)
// переобъявления функции abs
function abs(i1:integer returns integer)
function abs(i2:integer returns integer)
```

2.3. Объявления операций

Объявление операции выглядит как «*operation* знак операции (список формальных параметров *returns* список типов возвращаемых значений)», где число и типы формальных параметров и возвращаемых значений взаимозависимы со знаком операции и определяются в табл. 1 .

```
// допустимые объявления операций
operation : (complex returns integer)
operation < (complex, complex returns boolean)
operation . imag (complex returns real)
// недопустимые объявления операций
operation : (real returns integer)
operation < (complex, complex returns integer)
operation . imag (complex returns real, real)
```

Знак операции «:» задаёт операцию явного преобразования типов: «A_1 : R_1», где R_1 определяет тип, к которому приводится значение типа A_1. Тип R_1 разрешается заключать в квадратные скобки для устранения неоднозначной трактовки постфиксной операции точки и точки в имени типа, в случае неоднозначности трактуемой в пользу точки в имени типа.

Таблица 1

Допустимые объявления операций и их ограничения

<table>
<tr><th colspan="5">Знак операции</th><th>Вид операции</th><th>Ограничение</th></tr>
<tr><td colspan="5">:</td><td rowspan="2">[A_1 returns R_1]</td><td>Тип A_1 или тип R_1 должен быть пользовательским или инородным типом.</td></tr>
<tr><td colspan="5">ε [18]</td><td rowspan="3">Тип A_1 должен быть пользовательским или инородным типом.</td></tr>
<tr><td>+</td><td>-</td><td>!</td><td colspan="2">. имя</td><td rowspan="2">[A_1, …, A_n returns R_1, …, R_m]</td></tr>
<tr><td colspan="5">()</td></tr>
<tr><td colspan="5">[]</td><td>[A_1, A_2 returns R_1]</td></tr>
<tr><td colspan="2">=</td><td colspan="3"><</td><td>[A_1, A_2 returns boolean]</td><td rowspan="3">Тип A_1 или тип A_2 должен быть пользовательским или инородным типом.</td></tr>
<tr><td colspan="2">+</td><td colspan="3">-</td><td rowspan="2">[A_1, A_2 returns R_1]</td></tr>
<tr><td>*</td><td>/</td><td>%</td><td colspan="2">**</td></tr>
<tr><td>^</td><td>&</td><td>|</td><td colspan="2">||</td><td></td></tr>
</table>

```
interface complex_mod
  type complex := …
  operation : (integer returns complex)
end interface
module foo
  …
  // приведение integer типа к типу complex_mod . complex
  1 : complex_mod . complex
  …
  // то же, что и раньше, плюс взятие мнимой части
  1 : [ complex ] . imag
  …
end module
```

Знак операции «ε» задаёт неявное преобразование значения типа A_1 в значение тип R_1. Знак операции «!» задаёт унарную префиксную операцию: «! A_1». Операция со знаком «+» или «-» задаёт, в зависимости от количества формальных параметров, унарную префиксную операцию «знак операции A_1» или бинарную инфиксную операцию: «A_1 знак операции A_2». Знак операции «. имя» задаёт операцию «A_1 . имя». Знак операции «()» задаёт операцию «вызова функции»: «A_1 (A_2, …, A_n)». Знак операции «[]» задаёт операцию «доступа к элементу массива»: «A_1 [A_2]». Допустимо указывать последовательность индексов «A_1 [A_2, …, A_n]», которая разворачивается в последовательность применения операции «[]»: «A_1 [A_2] … [A_n]».

[18] Символ ε обозначает пустую цепочку, то есть объявление этой операции выглядит как «*operation* [A_1 *returns* R_1]».

```
type some1 := …
type some2 := …
operation [] (some1, integer returns some2)
operation [] (some2, real returns some2)
…
// данная операция допустима для значения A типа some1
A [ 1, 1.0, 2.0, 3.0 ]
```

Операция со знаком «=», «<», «*», «/», «%», «**», «^», «&», «|» или «||» задаёт бинарную инфиксную операцию: «A_1 знак операции A_2». Операция со знаком «=» через своё отрицание неявно задаёт операцию со знаком «!=». Операция со знаком «<» через своё отрицание неявно задаёт операцию со знаком «>=». Если одновременно заданы операции со знаками «=» и «<», то неявно определяются операции со знаками «>» и «<=».

В модуле нельзя объявлять две операции с одинаковым знаком, одинаковой формой и разной формой возвращаемых значений. В форму операций со знаками «:» и «ε» входит возвращаемый тип и, соответственно, форма возвращаемых значений этих операций пуста.

```
operation +(complex, complex returns complex)
// недопустимое объявление операции
operation +(complex, complex returns complex_record)
operation (complex returns integer)
// допустимое объявление операции
operation (complex returns real)
```

2.4. Определение контрактов

Контрактом называется набор процедур, в которых типы и параметры пользовательских типов с параметрами формальных параметров и возвращаемых значений могут быть заданы с помощью имён параметров контракта. Контракт определяется как «*contract* имя контракта [список имён параметров контракта[19]] объявления процедур *end contract*». Контракт не должен быть противоречив. Под противоречивым контрактом подразумевается контракт, в котором существует переопределение процедур в предположении, что имя параметра контракта задаёт тип, неэквивалентный другим типам.

```
contract need_add[T] // допустимый контракт
operation + (T, T returns T)
operation + (integer, integer returns integer)
function add (T, T, T returns T)
end contract
```

[19] Имена параметров должны быть различны.

```
contract any[T] // допустимый контракт без требований
end contract

contract need_add2[T] // недопустимый контракт
operation + (T, T returns T)
operation + (T, T returns integer)
end contract
```

Контракт может наследовать содержимое другого контракта, называемого базовым контрактом относительно определяемого контракта, следующим образом: «*contract* имя контракта [список имён параметров контракта] *of* имя базового контракта [список имён параметров базового контракта] объявления процедур *end contract*», где список имён параметров базового контракта входит в список имён параметров объявляемого контракта. Из определяемого контракта можно исключать объявления процедур базового контракта, предваряя их объявления ключевым словом «*no*».

```
contract need_add_mul[T] of contract need_add[T]
operation * (T, T returns T)
end contract

contract need_mul_div[T] of contract need_mul[T]
operation / (T, T returns T)
no operation + (T, T returns T)
no function add (T, T, T returns T)
end contract
```

Контракт приписывается объявлению и определению обобщенной процедуры и в её определении задаёт всё, что можно сделать с типами, заданными именами параметров контракта. В месте вызова обобщенной процедуры требуется наличие всех объявлений процедур (не являющихся обобщенными), указанных контрактом, после подстановки реальных типов вместо имён параметров контракта. Операции над строенными типами также могут быть использованы. После подстановки реальных типов вместо имён параметров контракта контракт может становиться противоречивым.

В одном модуле не могут быть определены разные контракты с одинаковым именем.

2.5. Объявления обобщенных процедур

В интерфейсе модуля можно объявить обобщенную процедуру, указав после имени функции и знака операции имя контракта и его параметров: «*of* имя контракта [список имён параметров контракта]». В типах формальных параметров и типах возвращаемых значений обобщенной процедуры допускается использование имён параметров контракта в качестве

типов и параметров пользовательских типов с параметрами. Параметры контракта, указываемые именами, не используемыми в типах формальных параметров обобщенной функции, называются свободными параметрами обобщенной процедуры. В модуле нельзя объявлять две обобщенные процедуры с одинаковым именем (или знаком операции), одинаковой формой возвращаемых значений (для функций) и разными именами контракта.

```
// определяем тип матрицы
type matrix[T] := array [..,..] of T
// определяем операцию умножения матриц
operation * of need_add_mul[T] (matrix[T], matrix[T] returns matrix[T])
// определяем операцию «замены» элемента матрицы
function insert of any[T](matrix[T], integer, integer, T returns matrix[T])
// переобъявление операции с другим контрактом недопустимо
operation * of need_mul_div[T] (matrix[T], matrix[T] returns matrix[T])
```

В обобщенных операциях допустимо использование пользовательских типов с параметрами вместо пользовательских типов там, где это требуется знаком операции. Обобщенные операции со свободными параметрами не разрешены. В модуле нельзя объявлять две обобщенные операции с одинаковым знаком, одинаковой формой и разной формой возвращаемых значений.

```
// обобщенные функции со свободными параметрами допустимы
function get_error of any[T] (returns T)
// обобщенные операции со свободными параметрами не разрешены
operation * of need_add_mul[T,T2] (matrix[T], matrix[T] returns matrix[T2])
// объявление операции недопустимо, так как уже была объявлена
// операция с такими типами формальных параметров
operation * of need_add_mul[T] (matrix[T], matrix[T] returns matrix[real])
```

Две формы обобщенной процедуры равны, если число их типов совпадает, а соответствующие типы эквивалентны между собой после эквивалентного переименования имён параметров контракта обеих форм. Тип, задаваемый именем параметра, считается эквивалентным только типу, задаваемому тем же именем параметра контракта. Под эквивалентным переименованием имён последовательности подразумевается последовательность имён, полученная после последовательного назначения новых имён, взятых из общей последовательности имён, при сохранении равенства равных имён.

```
function some of some[T1, T2, T3, T4] ( T1, T2, T2 returns T3, integer, T4)
// объявление той же обобщенной функции
function some of some[A, B, C, D] ( D, C, C returns B, integer, A)
// объявление других обобщенных функций
function some of some[A, B, C, D] ( D, C, D returns B, integer, A)
function some of some[A, B, C, D] ( D, C, C returns B, A, integer)
```

2.6. Импорт типов и контрактов

Импортировать все типы и контракты из других интерфейсов модулей можно конструкцией «*import* список имён интерфейса модуля». Можно импортировать конкретные типы или контракты по их имени конструкцией «*import* имя интерфейса модуля: объект импорта, …, объект импорта», где объектом импорта может быть тип «*type* имя типа» или контракт «*contract* имя контракта». Ключевые слова «*type*» и «*contract*» являются обязательными в случае наличия в импортируемом модуле объявления функции или одновременного наличия типа и контракта с таким же именем.

```
interface A
  type A1 = … type A2 = … type B1 := … type B2 := …
  contract A2 … end contract
  function B1 …
end interface

interface B
  import A
  // повторный импорт не ошибочен
  import A: A1, type A2, contract A2, type B1, B2
  // неоднозначный импорт ошибочен
  import A: A2
  // неоднозначный импорт ошибочен даже,
  // если функция B1 не может быть импортирована в интерфейс модуля
  import A: B1
```

Можно импортировать все типы и контракты модуля, кроме типов и контрактов, указанных конструкцией «*import* имя интерфейса модуля - объект импорта, …, объект импорта», например:

```
interface B
  // импортирует всё содержимое интерфейса модуля,
  // за исключением контракта A2 и типа B2
  import A - contract A2, B2
```

Импортируемые типы и контракты становятся частью интерфейса модуля, как если бы они были в нём определены или объявлены вместо конструкции *import*[20], за исключением того, что:

– при выборочном импорте типов неуказанные типы не импортируются, даже если они участвуют при построении импортируемых типов и для доступа к их содержимому (типам, лежащим в их основе) необходимо их импортировать[21];

[20] Таким образом отпадает необходимость использования оператора «.» в интерфейсе модуля.

[21] Данное требование, на первый взгляд, сводит объявление импорта типа до уровня его декларации (opaque type declaration в терминологии языка Sisal 2.0), однако импорт типа под-

– импортированный в интерфейсе модуля пользовательский тип и пользовательский тип с параметрами считается эквивалентным импортируемому типу.

```
interface A
  type someint := integer
  type someint2 := array of someint
end interface

interface B
  import A: type someint2
  // тип someint2 в модулях A и B эквивалентен и может быть использован
  function foo (someint2 returns null)
  // ошибка, тип someint не был импортирован
  function foo (someint returns null)
```

3. СТРУКТУРА МОДУЛЯ

Интерфейс модуля на языке Sisal начинается с ключевого слова *module*, за которым следует его имя. Модуль может содержать определения процедур, их объявления (предваряемые ключевым словом *forward*), определения и объявления типов, определения контрактов, а также конструкции импорта содержимого интерфейсов модулей. Содержимое модуля может разделяться точкой с запятой. Все объявления и определения влияют только на последующий текст модуля. Модуль завершается ключевыми словами «*end module*».

Например, далее приводится пример модуля «math», содержащего определения функций для вычисления факториала и числа Фибоначчи:

```
module math
  function fact (n: integer returns integer) // факториал числа n
    if n = 1 then 1 else fact(n-1)*n end if
  end function
  function fib (n: integer returns integer) // число Фибоначчи с номером n
    if n = 1 | n = 2 then 1 else fib(n-1) + fib(n-2) end if
  end function
end module
```

разумевает наличие определения типа (пусть и в интерфейсе другого модуля), что позволяет контролировать эквивалентность типов в модуле определения интерфейса и в модуле его использования.

3.1. Определение процедур

Определение процедуры выглядит как объявление процедуры (называемое заголовком определения), в котором должны быть указаны имена формальных параметров, и за которым следует список выражений[22]. Размерность списка выражений равна количеству возвращаемых значений, а типы значений можно неявно преобразовать[23] в соответствующие типы возвращаемых значений. Определение функции завершается ключевыми словами «*end function*». Определение операции завершается ключевыми словами «*end operation*». Определение процедуры также является соответствующим объявлением, если его не было сделано ранее.

```
function minmax1 (a: integer, b: integer returns integer, integer)
  if a < b then a, b else b, a end if
end function

// определение функции minmax2, эквивалентной функции minmax1
function minmax2 (a: integer, b: integer returns integer, integer)
  if a < b then a else b end if,
  if a < b then b else a end if
end function

// определение функции minmax3, использующее функцию minmax2,
// объявленную её определением
function minmax3 (a: integer, b: integer returns integer, integer)
  minmax2(a, b)
end function

// пример определения операции
operation - (c: complex returns complex)
  complex (-c.real, -c.imag)
end operation
```

Определение процедуры задаёт три области видимости объявлений: N_1, N_2 и N_3. Область видимости N_1 содержит объявления процедур, объявленных ранее. Область видимости N_2 не пуста только для определений обобщенных процедур, для которых она содержит объявления процедур контракта. Область видимости N_3 содержит имена формальных параметров определения процедуры, которые должны быть различными. Выражения процедуры могут определять последующие области видимости имён. Имена из областей видимости с большим номером перекрывают имена из областей видимости с меньшим номером. Также выражения могут использовать имена типов, объявленных и определённых ранее.

[22] Смотри раздел 5 для определения понятия списка выражений.

[23] Определение встроенных неявных преобразований типов находится в разделе 4.

```
// операция, возвращающая первые два значения,
// зависящие от определения операций контракта, и значение 3.0
function foo of need_add[T] (a: T, b: T returns T, integer, real)
  a + b, // используется операция контракта «operation + (T, T returns T)»
  let a := 1; b := 2 in
    a + b, // «operation + (integer, integer returns integer)» из контракта
    a:real + b // обычное сложение вещественного и целого значений, так как
               // операция из контракта неприменима,
              // ввиду отсутствия неявного преобразования real в integer
  end let
end function
```

Если у модуля в программе существует его интерфейс, то в модуле необходимо определить все процедуры, объявленные в его интерфейсе. Также в модуле необходимо определить все объявления процедур модуля. Определение функции сопоставляется её объявлению с той же формой формальных аргументов и возвращаемых значений. Определение операции сопоставляется её объявлению с той же формой формальных аргументов. Переопределение определений процедур не допускается.

```
// объявление и определение функции в модуле
forward function one (returns integer)
function one (returns integer) 1 end function
// объявление и определение различных функций
forward function neg (real returns real)
function neg (n: real returns real) -n end function

function nothing (returns null) nil end function
// переопределение функции nothing недопустимо
function nothing (returns null) nil end function
```

3.2. Импорт содержимого интерфейсов модулей

Конструкция импорта содержимого интерфейса модуля делает возможным использовать в модуле импортируемые процедуры, типы и контракты, в нём объявленные и определенные. Импортировать всё содержимое интерфейсов модулей можно конструкцией «*import* список имён интерфейса модуля».

Можно импортировать конкретные объявления или определения конструкцией «*import* имя интерфейса модуля: объект импорта, …, объект импорта». Объектом импорта может выступать:

- тип «*type* имя типа»;
- контракт «*contract* имя контракта»;
- конструкция «*function* имя функции [..]» для импорта всех объявлений функций с указанным именем;

- конструкция «*function* имя функции [список типов формальных параметров]» для импорта конкретной функции, однозначной по возвращаемым значениям;
- конструкция «*function* имя функции [список типов формальных параметров *returns* список типов возвращаемых значений]» для импорта конкретной функции;
- конструкция «*function* имя функции [список типов формальных параметров инородной функции]» для импорта инородной процедуры (инородная операция указывается её именем функции), где типы формальных параметров списка формальных параметров функции могут предваряться ключевыми словами «*raw*», «*in*», «*out*» и «*in out*»;
- конструкция «*function* имя функции *of* [имена параметров контракта] [список типов формальных параметров]» для импорта конкретной обобщённой функции, однозначной по возвращаемым значениям, где список типов формальных параметров зависит от имён указанных параметров контракта так же, как и у обобщенной функции, которую нужно указать;
- конструкция «*function* имя функции *of* [имена параметров контракта] [список типов формальных параметров *returns* список типов возвращаемых значений]» для импорта конкретной обобщённой функции;
- конструкция «*operation* знак операции [тип формального параметра *returns* тип возвращаемого значения]» для импорта операций со знаками «:» и «ε» и конструкция «*operation* знак операции [список типов формальных параметров]» для импорта операций с другими знаками;
- конструкция «*operation* знак операции *of* [имена параметров контракта] [тип формального параметра *returns* тип возвращаемого значения]» для импорта обобщённых операций со знаками «:» и «ε» и конструкция «*operation* знак операции *of* [имена параметров контракта] (список типов формальных параметров)» для импорта обобщённых операций с другими знаками.

Ключевые слова *type*, *contract* и *function* не обязательны, если в импортируемом модуле существуют только тип, контракт или функции с указанным именем. Ключевое слово *function* не обязательно, если после имени функции указана форма функции.

```
interface A
  type A1 = integer
  function A1 (integer returns integer)
  function A1 (integer returns integer, integer)
  function A1 (real returns integer)
  function A1 (integer, integer returns integer)
  function A2 (integer returns integer)
  function A2 (real returns real)
  contract any[T] end contract
  function A2 of any[T1, T2] (array of T1, T2 returns stream of T1)
  function A2 of any[T1, T2] (array of T1, T2 returns array of T1)
  function A2 of any[T1, T2] (array of T1, T1 returns stream of T1)
  operation (complex returns integer)
  contract need_intcast[T] operation (T returns integer) end contract
  operation of need_intcast[T] (matrix[T] returns matrix[integer])
  operation < (complex, complex returns boolean)
  contract need_cmp[T] operation < (T, T returns boolean) end contract
  operation < of need_cmp[T] (matrix[T], matrix[T] returns boolean)
end interface
module B
  // импортировать все объявления интерфейса модуля A
  import A
  // импортировать тип A1, контракт any и все функции с именами A1 и A2
  import A: type A1, any, function A1, A2
  // импортировать конкретные функции,
  // однозначные по возвращаемым значениям
  import A: A1 [real], function A1[integer], A1 [integer, integer]
  // импортировать конкретную функцию,
  // не обязательно однозначную по возвращаемым значениям
  import A: A1 [integer returns integer], A1 [integer returns integer]
  // импортировать обобщенную функцию,
  // однозначную по возвращаемым значениям (контракт any не  импортируется)
  import A: A2 of [T1, T2] [array of T1, T1]
  // импортировать обобщенную функцию,
  // не обязательно однозначную по возвращаемым значениям
  import A: A2 of [T1, T2] [array of T1, T2 returns stream of T1]
  // импортировать операции неявного преобразования типов
  import A: operation [complex returns integer]
  import A: operation of [T] [matrix[T] returns matrix[integer]]
  // импортировать операции сравнения
  import A: operation < [complex, complex]
  import A: operation < of [X] [complex[X], complex[X]]
```

Можно импортировать всё содержимое модуля, за исключением указанных объявлений и определений, конструкцией «*import* имя интерфейса модуля - объект импорта, …, объект импорта».

```
module B
  // импортировать всё, кроме функций с именем A2 и конкретной операции
  import A - A2, operation [complex returns integer], A2[integer]
```

При возникновении неоднозначности имени функции, типа или контракта, возникающей при наличии более одной функции, более одного типа или более одного контракта с одинаковым именем, принадлежащего разным модулям, перед неоднозначным именем обязательно указание имени модуля «имя модуля .»[24], которому принадлежит функция, тип или контракт. В одном модуле не может быть более одной операции с одинаковым знаком и формой[25].

```
interface A type T := integer end interface
interface B
  import A: type T
  function add (T, T returns T)
  operation + (T, T returns T)
end interface
interface C
  import A: T
  function add (T, T returns T)
  operation + (T, T returns T)
end interface
module D
  import A, B // импортировать интерфейс модуля A необязательно
  // импортировать интерфейс модуля C нельзя, так как из интерфейса модуля
  // B уже была импортирована операция «operation + (T, T returns T)»
  import C
  // эти предложения импорта правильные
  import C - operation + [T, T]
  import C: add [T, T]
  // в модуле D доступны две функции add: B.add и C.add
```

Если в программе существует интерфейс модуля **A**, то, неявно, его содержимое целиком импортируется в модуль **A**, как если бы конструкция импорта «*import* **A**» находилась сразу же после «*module* **A**».

```
interface some
  function foo(integer returns integer)
end interface
module some
  // функция foo была импортирована в модуль some неявным образом
  function foobar (i: integer returns integer) foo(i) end function
end module
```

[24] Данный префикс можно использовать не только для устранения неоднозначности имён.
[25] Таким образом, не возникает неоднозначности использования знака операции.

4. ТИПЫ И ОПЕРАЦИИ НАД НИМИ

Язык содержит следующие встроенные простые[26] типы: пустой (null), булевский (boolean), символьный (character), целый (integer) и вещественный (real). Встроенные простые типы не эквивалентны между собой. К встроенным составным типам[27] относятся потоки (*stream*), массивы (*array*), записи (*record*), объединения (*union*) и функции (*function*). Тип унарного выражения можно задать следующим образом: «*type* (унарное выражение)», например, «*type* (1.0+2)» равняется типу real.

Для каждого типа, кроме типа null и возможно инородных типов, существует выделенное ошибочное значение. Если не оговорено обратное, и аргумент операций над встроенными типами или аргумент предопределённых функций ошибочен, то результаты тоже будут ошибочными значениями. Узнать, ошибочно ли значение выражения, можно с помощью постфиксной операции «(выражение) *is error*».

4.1. Простые типы

Пустой тип состоит из единственного значения *nil*[28]. Никаких операций для пустого типа не определено.

Булевский тип состоит из значений истины (*true*), лжи (*false*) и ошибочного значения (*error*[boolean]). Для булевского типа определены бинарные[29] операции равенства («=»), неравенства («!=»), конъюнкции («&»), дизъюнкции («|»), исключающей дизъюнкции («^») и унарная[30] операция отрицания («!»). Определены операции явного преобразования между значениями булевского типа и значениями целого типа, так что значение *true* соответствует ненулевому числу, а значение *false* соответствует нулю.

```
// данные выражения истинны
! false, false = false, true = true, true != false, false != true,
true&true, true|false, false|true, true|true, true^false, false^true,
1 : boolean, true : integer = 1, false : integer = 0,
(error[boolean] = error[boolean]) is error

// данные выражения ложны
! true, true = false, false = true, true != true, false != false,
true&false, false&true, false&false, false|false, false^false, true^true,
0 : boolean
```

[26] Простыми называются типы, не определяемые через другие типы.
[27] Составными называются типы, определяемые через другие типы.
[28] Пустой тип используется для тегов объединения с неуказанными типами.
[29] Бинарные операции имеют два аргумента.
[30] Унарные операции имеют один аргумент.

Символьный тип содержит как минимум символы стандарта ASCII [12] и ошибочное значение (*error*[character]). Каждому символу соответствует уникальный неотрицательный целый код (для символов ASCII код описывается в соответствующем стандарте). Символы упорядочены в соответствии с порядком их кодов. Для символьного типа определены бинарные операции равенства, неравенства, больше («>»), меньше («<»), больше или равно («>=») и меньше или равно («<=»). Операция разности («-») двух символьных значений возвращает целое значение разности кодов символов.

```
// данные выражения истинны
'S' - 'S' = 0, '0' < '1' < '9' < 'A' < 'B' < 'Z' < 'a' < 'b' < 'z'
```

Определены операции явного преобразования между значениями символьного типа и значениями целого типа, так что символ соответствует числу кода этого символа. Литералы символьного типа имеют вид знака символа в одинарных кавычках. Также допускаются литералы, приведенные в табл. 2 , где ASCII-коды символов указаны как целые литералы языка Sisal.

```
'S', '\x53', '\83', '\o123' // все способы задания символа S
```

Таблица 2

Коды обратной косой черты (escape-последовательности)

Литерал	'\"'	'\\'	'\a'	'\b'	'\f'	'\n'	'\r'	'\t'	'\v'
Код	16#27	16#5C	16#7	16#8	16#C	16#A	16#D	16#9	16#B
Литерал	'\10-ричный код D'			'\o8-ричный код O'			'\h16-ричный код H'		
Код	10#10-ричный код D			8#8-ричный код O			16#16-ричный код H		

Целый тип содержит все числа достаточные для представления кодов символов символьного типа, их отрицания и ошибочное значение (*error*[integer]). Значения типа задаются цепочкой десятичных чисел, для повышения читаемости которой везде, кроме ее начала, могут использоваться символы подчеркивания. Знак числа, как и для вещественных чисел, считается унарной операцией, и поэтому между ним и числом допускается произвольное число пробельных символов. Целые литералы могут задаваться в произвольной системе счисления: «основание#число», где ее основание является числом от 2 до 36.

```
2#100_0000 = 8#100 = 16#40 = 36#1S = 64 // данное выражение истинно
```

Вещественный тип содержит значения, определяемые стандартом IEEE-754 [13], и ошибочное значение (*error*[real]). Значения вещественных типов

задаются десятичными литералами, отличающимися от целых литералов наличием точки и/или знаком экспоненты. Также для вещественных чисел символы подчеркивания не могут идти после десятичной точки. Вещественное число задается литералом простой или экспоненциальной формы. Простая форма является десятичным числом, разделенным знаком точки на две части, одна из которых может быть пуста. Экспоненциальная форма состоит из двух частей, разделенных буквой «e» или «E». Левая часть — это десятичное число, возможно разделяемое на две части точкой, которая может стоять вначале. Правая часть — это необязательный знак минуса или плюса и десятичное число.

```
5.0 = 5. = 0.5e1 = .5E1 = 50e-1 = 50_000.000E-4 // данное выражение истинно
```

Целый и вещественный типы имеют общее название числовых типов. Для числовых типов определены унарные операции идентичности («+»), смены знака («-»). Для числовых типов определены бинарные операции сложения («+»), вычитания («-»), умножения («*»), деления («/»), возведения в степень («**»), равенства, неравенства, больше, меньше, больше или равно и меньше или равно. Для целых значений определена бинарная операция взятия остатка от деления или модуля («%»). Определена операция неявного (и явного) преобразования целого значения в значение вещественного типа. Определена операция явного преобразования вещественного значения в значение целого типа «X : integer», возвращающая значение «floor (0.5 + X)».

```
// данные выражения истинны
+1 = 1, -1 = 0-1, 2+2 = 4, 2*3 = 6, 5/2 = 2, 5%2 = 1, 2**5 = 32,
+1.0 = 1.0, -1.0 = 0.0-1.0, 2.0*3.0 = 6.0, 5.0/2.0 = 2.5, 2.0**5.0 = 32.0,
1_000_000_000 : real = 1_000_000_000.0, // вероятна потеря точности
2.49 : integer = 2, 2.5 : integer = 3, 2.51 : integer = 3
```

Операция над целыми значениями порождает целое значение. При делении и возведении в степень целых значений берется целая часть результата. Операция над вещественными значениями порождает вещественное значение. Если операция над целыми значениями или операция явного преобразования вещественного значения в целое значение порождает число, не представимое типом integer, то порождается значение «*error*[integer]». Если в результате выполнения операции деления или модуля от целых операндов происходит деление на ноль, то возвращается значение «*error*[integer]». Для вещественного деления на ноль возвращается значение $\pm\infty$ стандарта IEEE-754.

```
// данные выражения истинны
2.0 + 2 = 4.0, 2.0:integer + 2 = 4, (1 / 0) is error, (1 % 0) is error
```

В интерфейсе модуля «std» определяются следующие функции:

```
// возвращает наибольшее целое число, не большее, чем вещественный аргумент
function floor (real returns integer)
// возвращает целое число, равное вещественному аргументу без дробной части
function trunc (real returns integer)
// возвращает модуль числа
function abs (integer returns integer)
function abs (real returns real)
// возвращает минимум двух чисел
function min (integer, integer returns integer)
function min (real, real returns real)
// возвращает максимум двух чисел
function max (integer, integer returns integer)
function max (real, real returns real)
```

4.2. Потоки

Тип потока описывается как «*stream of* тип элемента потока» и содержит возможно бесконечные цепочки последовательно доступных элементов одного типа и ошибочное значение «*error* [*stream of* тип элемента потока]». Типы потоков эквивалентны, если эквивалентны типы их элементов.

```
type si1 = stream of integer
// тип si1 эквивалентен типу si2
type integer2 = integer
type si2 = stream of integer2
```

Поток можно сконструировать в циклическом выражении[31] или с помощью выражения конструктора потока «*stream of* тип элемента потока [список значений элементов потока]» или «*stream* тип потока [список значений элементов потока]». Список значений элементов потока может отсутствовать для пустого потока. Для непустого списка значений элементов потока тип потока может отсутствовать и неявно задаваться типом первого элемента потока. Каждый элемент списка значений элемента потока последовательно задаёт одно или несколько значений элементов потока, начиная с элемента с индексом 1, и может быть выражением[32] или триплетом. Триплетом является структура вида «нижняя граница .. верхняя граница .. шаг», где все три элемента должны быть унарными выражениями целого типа и часто могут быть опущены[33] независимо друг от друга (шаг опускается вместе с точками «..», идущими впереди).

[31] Циклические выражения рассматриваются в разделе 5.5.
[32] Выражение задаёт количество элементов потока, равное его размерности.
[33] Триплеты используются не только в выражениях конструктора потока.

Триплет выражения конструктора потока задаёт арифметическую прогрессию элементов с указанной нижней границей и шагом, которые меньше или равны указанной верхней границы. Триплет задаёт по крайней мере один элемент (равный нижней границе). Если опущена нижняя граница, то она предполагается равной единице. Если опущена верхняя граница, то она предполагается равной бесконечности. Верхняя граница может быть опущена только в триплете, который является последним элементом списка значений элементов потока. Если опущен шаг, то он предполагается равным единице или минус единице, если нижняя граница больше верхней границы. Выражения, задающие конструкторы потоков, приведены ниже:

```
stream of integer [1..], // 1, 2, 3, …
stream si1 [1..4], // 1, 2, 3, 4
stream si2 [10..1..-3], // 10, 7, 4, 1
stream [..], // такой же поток, как и в первом выражении
stream [1..-3..1], stream of integer2 [], // пустой поток целых чисел
stream [], // пустой поток недопустим без указания его типа
stream of integer [0 .. .. -1], // 0, -1, -2, …
// поток вещественных чисел N-1, 1.0, 2.0, 3.0, 0.0, 0.0, 0.0
stream [(N-1):real, 1..3, 0, 0, 0],
// поток целых чисел N-1, 1, 2, 3, 0, 0, 0 (если N является целым числом)
stream [N-1, 1..3, 0, 0, 0]
```

Поток поддерживает выражение выбора элементов, задаваемое как «поток [выбирающее выражение]», где выбирающее выражение может быть унарным выражением целого типа (индексом), триплетом или унарным выражением типа целого потока или массива (индексным вектором). Выражение выбора, задаваемое индексом, возвращает выбираемый элемент. Выборка, задаваемая триплетом или индексным вектором, возвращает поток, составленный из выбираемых элементов. Если элемента с указанным индексом в потоке нет, то из потока выбирается ошибочное значение «*error* [тип элемента потока]». Все выражения выбора элементов потока являются более удобной записью совокупности выражений «поток [1]» (функция first) и «поток [2 ..]» (функция rest).

```
// выражение возвращает значения:
// stream [50, 60, 70, 80, 90, 100], 40, stream [70, 90] и 80
let  S := stream of integer [40, 50, 60, 70, 80, 90, 100 ]; N := 7
in S[2..], S[1], S[4..N-1..2], S[N-2]
end let,
// выражение S[4] являет сокращенной записью выражения, лежащего ниже:
let TS1 := S[2..]; TS2 := TS1[2..]; TS3 := TS2[2..] in TS3[1] end let
// выражение S[3..4] являет сокращенной записью выражения, лежащего ниже:
let TS1 := S[2..]; TS2 := TS1[2..]; V3 := TS2[1];
    TS3 := TS2[2..]; V4 := TS3[1]
in stream [V3, V4] end let
```

Для двух потоков определена бинарная операция конкатенации («||»), возвращающая поток, склеенный из двух указанных потоков. Программа неправильна, если типы потоков неэквивалентны, и не существует подходящей операции неявного преобразования типов элементов потоков[34].

```
// данное выражение истинно
let S1 := stream [10]; S2 := stream [20, 30];
    S3 := stream [40]; S4 := stream of integer []
in S1 || S2 || S3 || S4 end let = stream [10, 20, 30, 40]
```

Если тип элемента потока допускает префиксные и постфиксные[35] операции, то они допустимы и для этого потока, порождая поток с элементами, полученными после поэлементного применения операции над значениями исходного потока. Недопустимы постфиксные операции вызова функции более чем с одним результатом.

Если типы элементов двух потоков допускают инфиксные операции, то они допустимы и для этих потоков, порождая поток с элементами, полученными при поэлементном применении операции. Элементы меньшего по размеру потока дополняются ошибочными значениями.

Если тип элемента потока и значения, не являющегося потоком, допускают инфиксные операции, то они допустимы и для этого значения и потока, порождая поток с элементами, полученными при поэлементном применении операции для исходного потока и значения.

```
// выражение возвращает stream [false, true], stream [5, 7, 9],
// stream [8.0, 10.0, 12.0] и stream [1, 4, 27]
let S1 := stream [1, 2, 3]; S2 := stream [4, 5, 6];
    S3 := stream [true, false]
in !S3, S1 + S2, S2 * 2.0, S1 ** S1 end let
```

В интерфейсе модуля «std» определяются следующие функции и операции:

```
contract any[T] end contract // на тип T нет никаких ограничений
// возвращает true, если поток пуст, и false иначе
function empty of any[T] (stream of T returns boolean)
// возвращает поток, склеенный из двух указанных потоков
operation || of any[T] (stream of T, stream of T returns stream of T)
// следующие функции применяют соответствующие операции поэлементно
function floor (stream of real returns stream of integer)
function trunc (stream of real returns stream of integer)
function abs (stream of integer returns stream of integer)
```

[34] Сначала проверяется возможность неявного преобразования типа элемента второго потока к типу элемента первого потока, а потом возможность обратного преобразования.

[35] Кроме постфиксной операции квадратных скобок («[]»), так как она конфликтует с выражением выбора элементов потока.

```
function abs (stream of real returns stream of real)
function min (stream of integer, stream of integer
              returns stream of integer)
function min (stream of real, stream of real returns stream of real)
function max (stream of integer, stream of integer
              returns stream of integer)
function max (stream of real, stream of real returns stream of real)
```

4.3. Массивы

Тип массива описывается как «*array* форма массива *of* тип элемента массива» и содержит конечные цепочки элементов одного типа с прямым доступом по их многомерному индексу и ошибочное значение «*error* [*array* форма массива *of* тип элемента массива]». Форма может иметь свободный «[список двойных точек]» или фиксированный вид «[список дуплетов]». Дуплетом является структура вида «нижняя граница .. верхняя граница», где нижняя и верхняя границы являются унарными выражениями целого типа, чьи значения известны во время трансляции текущего модуля[36]. Нижняя граница может быть опущена и по умолчанию полагается равной единице. Верхняя граница должна быть больше или равна нижней границе. Форма может быть опущена и по умолчанию полагается равной «[..]». Количество размерностей массива задаётся размерностью формы массива, равной количеству элементов её списка двойных точек или дуплетов.

```
type Arr1 = array of integer // одномерный массив целых чисел
type Arr2 = array [..] of integer // одномерный массив целых чисел
type Arr3 = array [..,..] of integer // двухмерный массив целых чисел
type Arr4 = array [1..4] of integer // одномерный массив 4-х целых чисел
type Arr5 = array [1..5] of integer // одномерный массив пяти целых чисел
type Arr6 = array [..2,..3] of integer // 2x3 массив целых чисел
type Arr7 = array [2..5] of integer // одномерный массив 4-х целых чисел
type Arr8 = array [..3,..2] of integer // 3x2 массив целых чисел
```

Два типа массива эквивалентны, если эквивалентны типы их элементов и массивы имеют одинаковую форму. Формы массивов совпадают, если они имеют одинаковую размерность, они обе свободны или фиксированы, и у фиксированных форм совпадает количество элементов каждой размерности.

[36] Значение нижней и верхней границ массива не должно быть ошибочным значением.

```
// массив типа Arr1 эквивалентен массиву типа Arr2
// массив типа Arr4 эквивалентен массиву типа Arr7
// более никаких эквивалентностей среди типов массивов Arr1, …, Arr8 нет
```

Строковые литералы рассматриваются как одномерные массивы символов «*array of* character» (в интерфейсе модуля «std» находится определение типа «*type* string = *array of* character») с единичной нижней границей и задаются цепочкой символов, заключенной в двойные кавычки. Для задания символов строки допустимы все обозначения таблицы 2. Единственная особенность связана с синтаксисом задания символа его кодом: если за ним находится символ точки с запятой, то он считается маркером окончания кода и к строке не добавляется[37]. Если непосредственно перед начальной кавычкой находится символ «@», то все специальные обозначения символов, кроме цепочки «\"», воспринимаются буквально. Последовательные строковые литералы, возможно расположенные на разных строках, склеиваются в один.

```
"\83isal", "\83;isal", // строковой литерал "Sisal"
"foo" "bar", // один строковой литерал "foobar"
"foo" @"b\ar", // один строковой литерал "foob\\ar"
@"\foo" "bar", // один строковой литерал "\\foobar"
```

4.3.1. Выражение конструирования массива

Массив можно сконструировать в циклическом выражении или с помощью выражения конструктора массива. Выражение конструктора массива имеет вид «*array of* тип элемента массива [список значений элементов массива]», «*array* форма фиксированного вида *of* тип элемента массива [значения элементов массива]», «*array* форма фиксированного вида тип массива [значения элементов массива]» или «*array* тип массива с формой фиксированного вида [значения элементов массива]».

В указываемой форме необязательно использовать выражения, вычисляемые во время трансляции. В указываемой форме разрешается указывать перед дуплетом его имя «имя *in*», которое является уникальным для формы. Областью действий указанных имён является область значений элементов массива. Указываемая форма должна иметь количество размерностей, равное размерности указанного типа массива. В указываемой форме для типа массива с фиксированной формой можно указывать только имена нужных дуплетов, ставя двоеточие «..» для размерностей с не указанными именами.

[37] Тем самым возможно отделять код символа от идущих следом символов, которые могут быть проинтерпретированы как его часть.

Если форма конструируемого массива неизвестна, то конструируется одномерный массив с нижней границей равной единице и список значений элементов массива задаётся полностью аналогично конструктору потока. Для пустого массива список значений элементов массива может отсутствовать. Для непустого массива тип элемента массива может отсутствовать, возможно, вместе с ключевыми словами «*array of*» (для первого случая) и неявно задаваться типом первого указанного элемента массива.

```
// задаёт три одинаковых массива
array of integer [1, 2, 3], array of [1, 2, 3], [1, 2, 3]
```

Если форма массива задана, то значения элементов массива задаются непосредственно как «:= список значений элементов массива в row-major порядке»[38] или как список расположенных элементов, разделяемых точкой с запятой. Значения элементов массива задаются значениями типа **T**, эквивалентного типу элементов массива или неявно к нему приводимого.

```
// одинаковые одномерные массивы
array [1..3] of integer [:= 1, 2, 3], array [1..3] of [:= 1, 2, 3],
// одинаковые двухмерные массивы [[1, 2, 3], [4, 5, 6]]
array [1..2, 1..3] of integer [:= 1, 2, 3, 4, 5, 6],
array [..2, ..3] of [:= 1, 2, 3, 4, 5, 6],
// такой же массив, но с фиксированной формой
array array [1..2, 1..3] of integer [:= 1, 2, 3, 4, 5, 6]
```

Расположенные элементы задаются как «положение := расположенные значения элементов массива» или «*else* := расположенные значения элементов массива». Положение указывает задаваемые элементы массива и задается как список выражений целого типа или возможно именованных триплетов и индексных векторов. Если компонент положения ошибочен, или положение задаёт элементы вне массива, то указанные значения элементов массива не попадают в массив. Все незаданные элементы массива равны ошибочным значениям. Элементы, расположенные с помощью ключевого слова «*else*» (которое должно являться последним элементом списка расположенных элементов), соответствуют одному или более не заданных элементов массива[39]. Если в выражении конструктора массива описание расположения элементов пересекается, то возвращается ошибочный массив.

[38] При row-major порядке перечисления элементов массива элементы перечисляются, начиная с внутренней размерности массива. Далее именно такой порядок подразумевается в последовательности всех элементов массива.

[39] В выражении конструктора массива не может быть более одного описания расположения элементов с ключевым словом «*else*», и это описание не может быть единственным.

```
// массив [1, 2, 3]
array [1..3] of [1 := 1; 2 := 2; 3 := 3; 4 := 4],
array [1..3] of [1 := 1; 2 := 2; else := 3],
// массив array of real [1.0, 2.0, 3.0]
array [1..3] of [2 := 2.0; 1 := 1; 3 := 3],
// массив [[1, 0, 3], [4, 0, 6]]
array [1..2, 1..3] of [1,1 := 1; 1,3 := 3; 2,1 := 4; 2,3 := 6; else := 0],
// незаданные элементы массива равны ошибочным значениям
array [1..3] of integer [:= 1, 2], // массив [1, 2, error[integer]]
array [1..3] of [2 := 1; 3 := 2], // массив [error[integer], 1, 2]
array [1..3] of [1 := 1; 2 := 2; 3 := 3; 1 := 1] // ошибочный массив
```

Триплеты и индексные вектора могут разделяться ключевым словом «*dot*», а не запятой, и их последовательность называется последовательностью *dot*-элементов. Область действия имён триплетов и индексных векторов распространяется на последующие элементы списка выражений и расположенные значения элементов массива, если они заданы одним унарным выражением типа **T**.

Размерность списка положения должна быть равна размерности массива. Выражение задаёт одну или несколько размерностей списка в зависимости от своей размерности. Триплет и индексный вектор задают одну размерность списка. Каждая размерность списка положения задаёт индексы элементов, выбираемых из этой размерности. Неуказанная нижняя граница триплета равняется нижней границе соответствующей размерности массива. Неуказанная верхняя граница триплета равняется верхней границе соответствующей размерности массива. Неуказанный шаг триплета равняется единице.

Размерность формы положения определяется как размерность массива минус количество триплетов и индексных векторов плюс количество ключевых слов «*dot*». Каждой размерности формы положения соответствует триплет, индексный вектор или последовательность *dot*-элементов. Каждая размерность формы положения имеет нижнюю границу, равную нижней границе соответствующей размерности, и верхнюю границу, равную сумме нижней границы и количества выбираемых элементов соответствующей размерности. Последовательность *dot*-элементов имеет верхнюю границу, равную наибольшей размерности объединяемых триплетов и индексных векторов. Индексы последовательности *dot*-элементов изменяются вместе до тех пор, пока не исчерпаются индексы последнего элемента. Закончившиеся индексы равны значению «*error*[integer]».

Расположенные значения элементов массива могут задаваться непосредственно их перечислением в row-major порядке, унарным выражением типа **T** или массивом с размерностью формы положения и элементами типа **T**. Унарное выражение типа **T** задаёт все элементы, указанные положением.

Из массива с размерностью формы положения и элементами типа **T** выбираются элементы с индексами формы положения.

```
array [1..3] of [1..3 := 1, 2, 3], // массив[[1, 2, 3]
array [..2, ..3] of [..,.. := 1], // массив [[1, 1, 1], [1, 1, 1]]
// массив [[1, 2, 3], [4, 5, 6]]
array [..2, ..3] of [1,.. := 1, 2, 3; 2,.. := 4, 5, 6],
// массив [[1, 2, 3], [4, 5, 6]]
array [..2, ..3] of [1,.. := [1, 2, 3]; 2,.. := [4, 5, 6]],
// массив [[2, 0, 0], [0, 4, 0], [0, 0, 6]]
array [..3, ..3] of [i in .. dot j in .. := i+j; else := 0],
// массив [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
array [..3, ..3] of [.. dot .. := 1, 2, 3; else := 0],
// массив [[1, 1, 1], [0, 1, 1], [0, 0, 1]]
array [..3, ..3] of [i in .., i .. := 1; else := 0],
// массив [[0, 1, 2], [4, 5, 0]]
array [..2, ..3] of [1, [3, 2] := 1, 2; 2, [1, 2]. := 4, 5; else := 0]
```

4.3.2. Операции над массивами

Массив поддерживает выражение выбора и замещения элементов. Выражение выбора элементов массива имеет вид «массив [положение]», где положение описывалось в разделе 4.3.1. Выражение выбора элементов массива возвращает значение элемента массива, если размерность формы положения равна нулю. Если размерность формы положения не равна нулю, и количество элементов каждой размерности не зависит от других размерностей (форма массива задаёт прямоугольный массив), то возвращается массив с формой положения. Если размерность формы положения не равна нулю, и форма массива не задаёт прямоугольный массив, то возвращается массив массивов, каждый из которых соответствует последовательности размерностей формы, задающей прямоугольный массив.

```
// выражение let вычисляет значения:
// 3.0, [5, 6], [1.0, 4.0, 2.0, 1.0], true и true
let A := array [1..4] of real [1.0, 2.0, 3.0, 4.0];
    B := array [1..3, 2..4] of integer [1,.. := 1, 2, 3;
                                        2,.. := 4, 5, 6;
                                        3,.. := 7, 8, 9 ];
    U := array of integer [1, 4, 2, 1]
in A[3], B[2, 3..4], A[U],
   B [1..3 dot 4..2..-1] = array [3, 5, 7],
   B [3..2..-1 dot 2..4..2] = array [7, 6]
end let
```

Выражение замещения элементов массива имеет вид «массив [список расположенных элементов[40]]» и возвращает массив того же типа, что и у исходного массива и с теми же элементами, кроме заменяемых элементов. В списке расположенных элементов не разрешается использовать описание «*else*».

```
// выражение let вычисляет значения:
// [1, 2, 0, 4, 5], [1, 60, 70, 20, 10], [1, 2, 3, 4, 5]
let A := [1, 2, 3, 4, 5]
in A[3 := 0], A[2..5 := 60, 70, 20, 10], A end let,
// выражение let вычисляет значения:
// A и array [1..2, -3..-2] of [1,.. := 0, 2; 2,.. := 3, 0]
let A := array [1..2, -3..-2] of [1,.. := 1, 2; 2,.. := 3, 4]
in A, A[1..2 dot -3..-2 := 0] end let
```

Для двух массивов определена бинарная операция конкатенации («||»), возвращающая одномерный массив, склеенный из двух указанных массивов, элементы которых располагаются в row-major порядке. Программа неправильна, если типы массивов неэквивалентны и не существует подходящей операции неявного преобразования типов элементов массивов[41].

```
// данное выражение истинно
let A1 := [10]; A2 := array [..1, ..2] of [:= 20, 30];
    A3 := [40]; A4 := array of integer []
in A1 || A2 || A3 || A4 end let = [10, 20, 30, 40]
```

Если тип элемента массива допускает префиксные и постфиксные[42] операции, то они допустимы и для этого массива, при этом порождается массив с нижней границей как у исходного и элементами, полученными после поэлементного применения операции над значениями исходного массива. Не допустимы постфиксные операции вызова функции более чем с одним результатом.

Если типы элементов двух массивов допускают инфиксные операции, то они допустимы и для этих массивов. При этом порождается массив с нижней границей, равной единице, и элементами, полученными при поэлементном применении операции без учета нижних границ, причем элементы меньшего по размеру массива дополняются ошибочными значениями. Определена операция неявного преобразования типа массива к типу потока, располагающая элемент массива в row-major порядке.

---

[40] Список расположенных элементов описывался в разделе 4.3.1.

[41] Сначала проверяется возможность неявного преобразования типа элемента второго массива к типу элемента первого массива, а потом возможность обратного преобразования.

[42] Кроме постфиксной операции квадратных скобок («[]»), так как она конфликтует с выражением выбора и замещения элементов массива.

Если тип элемента массива и значения допускают инфиксные операции, то они допустимы для этого значения и массива, порождая массив с нижней границей исходного массива и элементами, полученными при поэлементном применении операции для исходного массива и значения.

```
// выражение let вычисляет значения [false, true], [2, 4, 6] и
// array [1..2, 1..2] of [:= 8.0, 10.0, 12.0, 14.0]
let A1 := [1, 2, 3];
    A2 := array [5..6, -1..0] of [5,.. := 4, 5; 6,.. := 6, 7];
    A3 := [true, false]
in ~A3, A1 + A1, A2 * 2.0 end let
```

Для массива определяется функция одного аргумента «size», которая берёт на входе массив и возвращает количество элементов во всех его размерностях. Функция «size» определена для двух элементов и возвращает количество элементов его размерности, указанной вторым аргументом целого типа. Для массива определяется функция одного аргумента «transpose», которая берёт на входе массив и возвращает транспонированный массив.

Для массива определяется функция одного аргумента «liml», которая берёт на входе массив и возвращает нижнюю границу его первой размерности. Функция «liml» определена для двух элементов и возвращает нижнюю границу его размерности, указанной вторым аргументом целого типа. Для массива определяется функция одного аргумента «limh», которая берёт на входе массив и возвращает верхнюю границу его первой размерности. Функция «limh» определена для двух элементов и возвращает верхнюю границу его размерности, указанной вторым аргументом целого типа.

Для массива определены функции «floor», «trunc», «abs», «min» и «max», выполняющие соответствующие операции поэлементно.

4.4. Записи

Тип записи «*record* [объявление полей записи]» задает декартово произведение типов своих полей, к значениям которых имеется прямой доступ по их уникальному в пределах одного типа записи имени. Объявление полей записи содержит разделяемые точкой с запятой объявления полей с одинаковым типом «список имён полей : тип полей». Тип записи содержит ошибочное значение. Если запись ошибочна, то значение всех её полей тоже ошибочно. Типы записей эквивалентны, если они имеют одинаковое количество полей и типы соответствующих полей эквивалентны.

```
// рекурсивное определение записи, в отличие от рекурсивного
// определения объединения, не допустимо
```

```
type bad_stack := record [ value: real; rest: bad_stack]
// имя типа записи может быть использовано в качестве имени её поля
type Ex1 = record [Ex1: record [ Ex1: real ]]
// имя поля записи может быть использовано в другой записи
type Ex2 = record [Ex1 : Ex1]
// типы записей XYrec и YXrec не эквивалентны
type XYrec = record [ X: real; Y: integer ]
type YXrec = record [ X: integer; Y: real ]
// типы записей XYrec и ABrec эквивалентны
type ABrec = record [ A: real; B: integer ]
```

Запись конструируется как «*record* тип записи [определение полей записи]», «*record* [определение полей записи]» или «*record* тип записи [:= список выражений]». Определение полей записи содержит разделяемые точкой с запятой определения одного или нескольких полей «список имён полей := список выражений», указанных их именами. Для задания полей записи, являющейся полем записи, в качестве имени поля можно использовать несколько имён полей, разделённых точкой. Список выражений должен определять все указанные поля записи, а в выражении конструктора записи единственным образом должны быть определены все её поля. Значение выражений может неявно приводиться к типам полей, указанных типом записи.

```
// различные способы построения одной записи
record XYrec [X:= 2.0; Y: 1], record XYrec [Y: 1; X:= 2.0],
record XYrec [X, Y := 2.0, 1], record XYrec [Y, X := 1, 2.0],
record XYrec [:= 2.0, 1],
// построение записи с указанием вложенных полей
record Ex1 [Ex1.Ex1 := 1.0],
// построение записи record [a: real; b: integer] «на месте»
record [a:= 1.0; b := 1]
```

Значение поля записи можно получить как «запись . имя поля этой записи». Определена операция «замещения» полей записи «запись *replace* [определение полей записи]», которая создает новую запись такого же типа, но с новыми значениями полей, указанных их именами. Если запись является ошибочным значением, то порождается также ошибочное значение.

```
// получение значение поля записи
record XYrec [Y, X := 1, 2.0] . X, // выражение равно 2.0
// выражение равно record [ Ex1 := 1.0 ]
record Ex1 [Ex1.Ex1 := 1.0] . Ex1,
record Ex1 [Ex1.Ex1 := 1.0] . Ex1 . Ex1, // выражение равно 1.0
// выражение равно record XYrec [:= 1.0, 1]
record XYrec [:= 2.0, 1] replace [X := 1.0],
// выражение равно record Ex1 [Ex1.Ex1 := 2.0]
record Ex1 [Ex1.Ex1 := 1.0] replace [Ex1.Ex1 := 2.0]
```

4.5. Объединения

Тип объединения «*union* [объявление тегов объединения]» аналогичен типу записи, исключая то, что не более чем одно значение его тега может быть отлично от ошибочного значения. Объявление тегов объединения содержит разделяемые точкой с запятой объявления тегов с одинаковым типом «список имён тегов : тип тегов». При объявлении тегов объединения можно опускать их типы (вместе с двоеточием), если они равны null. Тип объединения содержит ошибочное значение. Если объединение ошибочно, то значение всех его тегов тоже ошибочно. Типы объединений эквивалентны, если они имеют одинаковое количество тегов и типы соответствующих тегов эквивалентны.

```
// примеры определения объединений
type UnEx1 = union [ T1: real;    T2: integer ]
type UnEx2 = union [ T1: integer; T2: UnEx1 ]
type StNode := union [ Empty; Element: record [Value: real; Next: StNode] ]
type UnType := union [ red, green, blue, black, white ]
```

Объединение конструируется как «*union* тип объединения [имя тега этого объединения := значение тега]», где значение тега вместе с со знаком присваивания можно опускать, если тип тега равен типу null. Значение тега может неявно приводиться к типу тега объединения. Выражение «объединение *is tag* имя тега этого объединения» истинно, если имя тега равно имени тега данного объединения; ложно, если не равно и ошибочно, если само объединение ошибочно. Конструкция «объединение . имя тега этого объединения» равна значению, заданному указанным именем тега объединения.

```
// примеры конструирования объединений
union UnEx2 [ T1 := 1 ],
union UnEx2 [ T2 := union UnEx1 [ T1 := 2.0 ] ],
union UnType [ red ],
// примеры проверки тегов объединений
union UnEx1 [ T2 := 1] is tag T2, // выражение равняется true
union UnEx1 [ T2 := 1] is tag T1, // выражение равняется false
// примеры получения значений тегов объединений
union UnEx1 [ T2:= 1] . T2, // выражение равняется 1
union UnEx1 [ T2:= 1] . T1 // выражение равняется error[real]
```

4.6. Функции

Тип функции задается как «*function* [типы аргументов *returns* типы результатов]», содержит все процедуры с указанными типами аргументов и результатов и ошибочное значение. Значение типа функции можно сконструировать с помощью:

- имени объявленной необобщенной функции «имя функции»[43], если имя функции и имя модуля не перекрыто локальным именем и функция задаётся однозначно[44];
- имени функции «*function* имя функции [..]», позволяющего указать однозначно заданную необобщенную функцию, даже если она и имя её модуля перекрыто локальным именем;
- имени обобщенной функции «имя функции . [типы свободных параметров]» только со свободными параметрами, которая задана однозначно (даже если она и имя её модуля перекрыто локальным именем);
- конструкции «*function* имя функции [список типов формальных параметров]» для указания конкретной необобщенной функции однозначной по возвращаемым значениям или, в случае её отсутствия, максимально простой[45] обобщенной функции без свободных параметров и однозначной по возвращаемым значениям, в которой значения параметров восстанавливаются путём прохода по типам формальных параметров слева направо;
- конструкции «*function* имя функции [список типов формальных параметров *returns* список типов возвращаемых значений]» для указания конкретной необобщенной функции или, в случае её отсутствия, максимально простой обобщенной функции;
- конструкции «*function* имя функции [список типов формальных параметров инородной функции]» для указания инородной процедуры (инородная операция указывается её именем функции)[46], где типы формальных параметров списка формальных параметров функции могут предваряться ключевыми словами «*raw*», «*in*», «*out*» и «*in out*»;

---

[43] Имя функции может всегда указываться с помощью имени модуля «имя модуля . имя функции».

[44] Операция вызова функции позволяет задавать обобщенную функцию с помощью неоднозначного имени.

[45] Максимальной простой обобщенной функцией среди всех других обобщенных функций с тем же именем, которые можно применить при определённом обозначении параметров, является та, которая содержит минимальное число несвободных параметров.

[46] Инородной функции соответствует тип функции, полученный после добавления возвращаемых значений для формальных параметров с ключевыми словами «*in out*» и «*out*», удаления формальных параметров только с ключевым словом «*out*» и удаления всех ключевых слов «*raw*», «*in*», «*out*» и «*in out*».

- конструкции «*operation* знак операции [тип формального параметра *returns* тип возвращаемого значения]» для указания операций[47] со знаками «:» и «ε» и конструкции «*operation* знак операции [список типов формальных параметров]» для указания операций с другими знаками;
- конструкции «*function* имя функции . [типы свободных параметров] [список типов формальных параметров]» для указания максимально простой обобщённой функции со свободными параметрами, однозначной по возвращаемым значениям;
- конструкции «*function* имя функции *of* [имена и типы параметров контракта] [список типов формальных параметров]» для указания конкретной обобщённой функции, однозначной по возвращаемым значениям, где имена и типы параметров контракта содержат перечисляемые через запятую структуры вида «имя = тип», которые можно сокращать до «тип» для свободных параметров, а список типов формальных параметров зависит от имён указанных параметров контракта так же, как и у обобщенной функции, которую нужно указать;
- конструкции «*function* имя функции *of* [имена и типы параметров контракта] [список типов формальных параметров *returns* список типов возвращаемых значений]» для указания конкретной обобщённой функции;
- конструкции «*operation* знак операции *of* [имена и типы параметров контракта] [тип формального параметра *returns* тип возвращаемого значения]» для указания конкретных обобщённых операций со знаками «:» и «ε» и конструкции «*operation* знак операции *of* [имена и типы параметров контракта] (список типов формальных параметров)» для импорта обобщённых операций с другими знаками;
- конструкции «*function* имя функции (список формальных параметров *returns* типы результатов) список выражений *end function*» для определения λ-функции, в которой имя функции необязательно и используется в списке выражений функции только для задания рекурсивной зависимости (в списке выражений функции разрешается использовать имена значений, определённых вне тела функции).

[47] Обобщенная операция может быть указана таким образом, если не существует подходящей необобщенной операции и существует максимально подходящая обобщенная функция.

```
type tfoo1 = function [integer returns integer]
type tfoo2 = function [real returns real]
type tfoo3 = function [integer returns integer, integer]
type tfoo4 = function [integer, integer returns integer, integer]
type tfoo5 = function [returns integer]

function foo1 (i: integer returns integer) i end function
function foo2 (i: integer returns integer) i end function
function foo2 (r: real returns real) r end function
function foo3 (i: integer returns integer) i end function
function foo3 (i: integer returns integer, integer) i, i end function
function foo4 of any[T] (t: T returns T) t end function
function foo5 of any[T] (t: T returns T) t end function
function foo5 of any[T] (t1: T, t2: T returns T, T) t1, t2 end function
function foo6 of any[T] (t: T returns T) t end function
function foo6 of any[T] (t: T returns T, T) t, t end function
function foo7 of any[T] (returns T) error[T] end function
function foo8 of any[T] (returns T) error[T] end function
function foo8 of any[T] (integer returns T) error[T] end function

function test1 (foo1: integer returns integer, tfoo1, tfoo2, tfoo3,
                                       tfoo1, tfoo4, tfoo1, tfoo5,
                                       tfoo5, tfoo1, tfoo1, tfoo1)
  foo1,
  function foo1 [..],
  function foo2 [real],
  function foo3 [integer returns integer, integer],
  function foo4 of [T=integer] [T],
  function foo5 of [T=integer] [T, T],
  function foo6 of [T=integer] [T returns T],
  function foo7 . [integer] [],
  foo7 . [integer],
  function foo8 . [integer] [integer],
  function (i: integer returns integer) i + foo1 end function,
  function fact (n: integer returns integer)
    if n = 1 then 1 else fact(n-1)*n end if
  end function
end function
```

Тип функции имеет операцию её вызова «функция (значения аргументов)», возвращающую результаты функции, вычисленные после подстановки значений аргументов на место имён формальных параметров. Если значение функции ошибочно, то результаты операции её вызова будут ошибочны.

```
function test2 (returns integer, real, integer)
  foo2(1), foo2(1.0), foo1(1.0)
end function
```

Операция вызова функции автоматически разрешает неоднозначность функции, заданной её именем, на основании типов ее аргументов (если функция однозначна по возвращаемым значениям). Если не существует функции с формой, состоящей из типов, эквивалентных типам аргументов, то выбирается функция с минимальным количеством[48] неявных преобразований, которое необходимо осуществить для преобразования типов значений аргументов в типы формальных параметров.

Если не существует подходящей необобщенной функции, и функция задана своим именем, то операция вызова функции может указать максимально подходящую обобщенную функцию без свободных параметров. Параметры обобщенных функций восстанавливаются путём прохода по типам формальных параметров слева направо с попытками применения неявных преобразований типов значений аргументов в типы формальных параметров. Если отсутствует обобщенная функция, требующая минимального количества неявных преобразований, то среди обобщенных функций с наименьшем количеством неявных преобразований выбирается максимально простая обобщенная функция.

```
function test3 (returns real)
  foo5(1.0, 1) // вызывается функция «function foo5 of [T=real] [T, T]»
end function
```

Однозначно заданную обобщенную функцию со свободными параметрами можно вызвать следующим образом: «имя функции . [типы свободных параметров] (значения аргументов)». Типы свободных параметров указываются в порядке их перечисления в контракте объявления обобщенной функции[49]. В остальном вызов обобщенной функции со свободными параметрами аналогичен вызову обобщенной функции без свободных параметров.

```
function test4 (returns real)
  // вызывается функция «function foo8 of [T=real] [integer returns T]»
  foo8.[real](1)
end function
```

[48] С учётом количества неявных преобразований, полученных при применении свойства дистрибутивности неявного преобразования.

[49] Порядок следования типов свободных параметров гарантированно сохраняется при переобъявлении обобщенной функции.

Если аргумент функции пропущен, то результатом операции вызова функции, называемой теперь «сужением» области определения функции, будет функция от пропущенных аргументов в порядке их следования и с результатами исходной функции. Если значение функции ошибочно, то операция её сужения возвратит ошибочное значение функции. Операция сужения функции также разрешает неоднозначность функции на основании типов её аргументов, указываемых как «: тип» для пропускаемых аргументов.

```
function test5 (returns function [integer returns integer, integer],
                function [real returns real, real])
  foo5(1,), foo5(:real,1)
end function
```

4.7. Пользовательские типы

Пользовательские типы отличаются от всех прочих тем, что их необходимо определять от некоторого базового типа, который тоже в свою очередь может быть пользовательским типом. Пользовательский тип **A** основывается на другом пользовательском типе **B**, если тип **B** является базовым типом типа **A**, или тип **B** является базовым типом пользовательского типа, на котором основывается тип **A**. Встроенный тип **B** лежит в основе пользовательского типа **A**, если он является базовым типом типа **A**, или он является базовым пользовательского типа, на котором основывается тип **A**. Пользовательский тип является подмножеством значений своего базового типа.

```
// пример пользовательского типа, поддерживающего сумму двух чисел
type sum_pair := record [ a, b, sum: integer ]
```

Для пользовательского типа не определяется никаких встроенных операций[50], кроме как операций явного преобразования базового типа к пользовательскому типу и операции явного преобразования пользовательского типа к пользовательскому типу, на котором он основывается, и к встроенному типу, лежащему в его основе. Непереопределяемые операции явного преобразования пользовательского типа к пользовательскому типу, на котором он основывается, и к встроенному типу, лежащему в его основе, возвращают значение, из которого данное значение пользовательского типа было сконструировано. Операцию явного преобразования базового типа к

[50] Операции базового типа пользовательским типом не наследуются, и для их использования необходимо выполнить преобразование к базовому типу.

пользовательскому типу можно переопределить[51], и в её определении происходит неявное преобразование значения базового типа в значение пользовательского типа.

```
// конструктор, гарантирующий правильное состояние поля sum типа sum_pair
operation : (base: record [ a, b, sum: integer ] returns sum_pair)
  base replace [sum := base.a + base.b]
end operation
```

Ошибочное значение пользовательского типа конструируется путём применения операций явных преобразований типов к ошибочному значению типа, лежащего в основе пользовательского типа.

4.8. Инородные типы

Инородной тип задаётся строкой «“строковое представление инородного типа”», где строковое представление инородного типа должно задаваться на языке Си++ для использования в инородных функциях на языке Си++ и на подмножестве языка Си++, соответствующего языку Си, для инородных функций на языке Си и Фортран[52]. Инородные типы эквивалентны, только если совпадает их строковое представление, что является более строгим критерием эквивалентности, чем это необходимо, и в обязанности программиста входит поддержание его правильности.

```
// типы int1 и int2 считаются разными в языке Sisal,
// но в языке Си это не так
type int1 = "long"
type int2 = "long int"
```

Никаких операций для инородного типа изначально не определено. Значения инородных типов конструируются только в инородных процедурах. Если для инородного типа **T** определена операция «*operation* (**T** *returns* **T**)», то она используется для создания копии значения инородного типа **T**. Если операция копирования запрещена («*no operation* (**T** *returns* **T**)»), то копирование значения инородного типа тоже запрещено. Если операция копирования не определена и не запрещена, то используется побитовое копирование значения инородного типа.

---

[51] Переопределение операции явного преобразования базового типа к пользовательскому типу позволяет задать ограничения на содержимое (правильность) базового типа, которое невозможно обойти при создании пользовательского типа.

[52] Вызов функций и процедур языка Фортран возможен только для процедур, использующих средства взаимодействия с языком Си, появившихся в языке Фортран-2003.

Если для инородного типа **T** определена операция «*operation* (**T** *returns* null)», то она используется для освобождения копии значения инородного типа **T**, иначе никаких специальных действий для освобождения копии инородного типа не выполняется. Ошибочное значение инородного типа **T** соответствует его неопределенному значению, если не определена операция «*operation* (null *returns* **T**)», которая возвращает значение инородного типа **T**, соответствующее ошибочному значению инородного типа.

```
// строки в динамической памяти, оканчивающиеся нулевым символом
type psz := "char*"
operation (array of character returns psz)
operation (psz returns array of character)
operation (psz returns psz)  // копирование строк возможно
operation (psz returns null)
operation (null returns psz) // error[psz] это нулевой указатель
```

5. ВЫРАЖЕНИЯ

Выражения языка Sisal могут быть n-арными. Любое унарное (n=1) выражение языка Sisal рассматривается как арифметическое выражение. Список выражений задаётся перечисленными через запятую выражениями. Размерность списка выражений равна сумме размерностей выражений, в него входящих.

```
// размерность данного списка выражений равна трём
if a < b then a, b else b, a end if, 1
```

Перед каждым выражением могут располагаться прагмы[53] «assert = булевское условие», которые могут проверяться на истинность компилятором после вычисления выражения и использоваться при оптимизирующих преобразованиях программы. Результат унарного выражения в булевском выражении обозначается через одиночный символ подчеркивания «_». Арности n-арного (n≥1) выражения обозначаются как «_[1]», …, «_[n]».

Прагмы «pre_assert = булевское условие» и «post_assert = булевское условие» могут располагаться перед ключевым словом «*returns*» в объявлениях процедур и накладывать условия на возвращаемые значения и указанные имена формальных параметров, проверяемые до (pre_assert) или после (post_assert) вызова процедуры.

[53] Подробнее о прагмах можно прочитать в разделе 7.3.

```
// факториал числа n
forward function fact (n: integer
/*$ assert = n >= 1*/ /*$ assert = _ >= n*/
  returns integer)

function fact (n: integer returns integer)
  if n = 1 then 1 else /*$ assert = _ > 0*/ fact(n-1)*n end if
end function
```

5.1. Арифметическое выражение

Арифметическое выражение содержит операнды и операции. Операндами являются унарные выражения. Операции могут быть постфиксными, префиксными и инфиксными.

Постфиксные операции имеют вид «операнд операция». Цепочка постфиксных операций вычисляется слева направо до начала вычисления префиксных операций. К постфиксным операциям относятся операции вызова и «сужения» функции, выбора и замены элементов массива, выбора элементов потока, доступа к полю записи или объединения, замены элементов записи, проверка тега объединения и операция явного приведения типов.

Префиксные (унарные) операции имеют вид «знак операции операнд». Цепочка префиксных операций вычисляется справа налево до начала вычисления инфиксных операций. К префиксным операциям относятся операции смены знака («-»), идентичности («+») и логического отрицания («!»).

Инфиксные (бинарные) операции имеют вид «операнд знак операнд». Среди нескольких инфиксных операций раньше выполняются операции на более глубоком уровне вложенности арифметических скобок. Среди инфиксных операций одного уровня вложенности сначала выполняются операции с большим приоритетом, указанным в таблице 3 . Лево-связываемые операции одного приоритета выполняются слева направо, а право-связываемые операции — справа налево. Цепочка операций сравнения объединяется операциями конкатенации, например «A < B <= C» рассматривается как «A < B & B <= C».

```
(1+(3+5)*10)/3 ** 2 // в результате получаем число 9
```

Таблица 3

Свойства инфиксных операций

Приоритет	1	2	3	4	5	6	7	8	9
Знак	||	|	^	&	= !=	< > <= >=	- +	* / %	**
Связывание	«Левое»								«Правое»

Если для операндов инфиксной операции не объявлена операция для типов, эквивалентных типам операндов, то делаются попытки неявных преобразований и применений операции над получившимися типами. Сначала делается попытка неявного преобразования второго операнда к типу первого операнда, а потом попытка неявного преобразования первого операнда к типу второго операнда.

Операция неявного преобразования типов дистрибутивна. Если определена операция неявного преобразования типа **A** в тип **B** и операция неявного преобразования из типа **B** в тип **C**, то определена операция неявного преобразования типа **A** в тип **C** через тип **B**. Действует правило применения минимального неявного преобразования, которое вызовет непосредственно преобразование из типа **A** в тип **C**, если оно определено.

5.2. Выражение «let»

Выражение «*let*» определяет новую область и множество ее имен, используя их для вычисления списка выражений своих результатов: «*let* определения имен *in* список выражений результатов *end let*». Определения имен содержат разделенные точкой с запятой определения, содержащие левую и правую части, разделенные символами знака равенства с двоеточием. Левая часть — это разделенные запятыми определяемые имена, после каждого из которых может явно указываться его «: тип». Правая часть состоит из списка выражений, сумма размерностей которых равна числу имен левой части определения. Выражения правой части определения не могут зависеть от имен его левой части. Область действия определённых имён состоит из правых частей последующих определений и списка выражений результатов.

```
// выражение равно 3.0 * G * 3.0, 3.0
let X := 3.0; A := X * G in A * X, X end let
// данное выражение равно 4
let A := 3 in let A := A + 1 in A end let end let
```

5.3. Выражение «if»

Выражение «*if*» выглядит как «*if* булевское условие *then* список выражений результата ветви *elseif* ветвь *else* *end if*», где каждая из необязательных ветвей «*elseif*» задаётся как «*elseif* булевское условие *then* список выражений результата», а необязательная ветвь «*else*» задаётся как «*else* список выражений результата».

У всех списков выражений результата размерности должны быть равны. Типы возвращаемых значений выражения «*if*» определяются типами размерностей в первом списке выражений результата (после ключевого «*then*»). Типы размерностей в других списках выражений результата должны быть эквивалентны типам соответствующих результатов выражения «*if*» или неявно к ним приводиться.

Выражения булевских условий вычисляются последовательно, пока не получится истинное значение. Список выражений результата, идущий за первым истинным булевским условием, определяет результаты выражения «*if*». Если все булевские условия ложны, то ветвь «*else*» определяет результат конструкции. Если ветвь «*else*» отсутствует или встретилось ошибочное значение булевского условия, то выражение «*if*» возвращает ошибочные значения.

Ниже приведено простое выражение «*if*», описывающее модуль числа x:

```
if x < 0 then -x else x end if
```

Далее приведено более сложное выражение «*if*», вычисляющее корни квадратного уравнения в зависимости от знака дискриминанта:

```
let d := b**2 - 4*a*c
in if    d > 0 then (-b+d**0.5)/2*a, (-b-d**0.5)/2*a
  elseif d = 0 then          -b/2*a,          -b/2*a
  else                  error[real],     error[real]
  end if
end let
```

5.4. Выражение «case»

Выражение «*case*» выглядит как «*case* управляющее выражение условные ветви ветвь *else end case*», где должна присутствовать хотя бы одна условная ветвь вида «*of* список значений тестов *then* список выражений результата», а необязательная ветвь «*else*» задаётся как «*else* список выражений результата». Управляющее выражение должно быть унарным.

У всех списков выражений результата размерности должны быть равны. Типы возвращаемых значений выражения «*case*» определяются типами размерностей в первом списке выражений результата. Типы размерностей в других списках выражений результата должны быть эквивалентны типам соответствующих результатов выражения «*case*» или неявно к ним приводиться.

Значение теста может быть унарным выражением и, если тип управляющего выражения имеет операции со знаками «=» и «<», дуплетом «нижняя граница .. верхняя граница», в котором опущенные унарные выраже-

ния нижней и верхней границы по умолчанию равняются минус и плюс бесконечности. Если значение теста является унарным выражением, то тестом является сравнение его значения со значением управляющего выражения. Если значением теста является дуплет, то тестом является булевское выражение «нижняя граница <= управляющее выражение <= верхняя граница».

Значение управляющего выражения проверяется тестом каждой условной ветви в порядке их следования до первого совпадения. Список выражений результата, идущий за первым истинным тестом, определяет результаты выражения «*case*». Если все тесты ложны, то ветвь «*else*» определяет результат конструкции. Если ветвь «*else*» отсутствует или значение теста ошибочно, то выражение «*case*» возвращает ошибочные значения.

```
case die_1 + die_2
  of 2..3,    12 then        "lose"
  of    7,    11 then         "win"
  of 4..6, 8..10 then "no decision"
  else error[array of character]
end case
```

Если известно, что истинным может быть только один тест, то для выражения «*case*» можно указать прагму «parallel» или прагму «parallel = булевское условие», если тест может быть истинным при определённом булевском условии.

Для выбора условий по тегам объединения существует выражение «*case tag*», выглядящее как «*case tag* управляющее выражение типа объединение условные ветви выражения *case tag* ветвь *else end case*». Значениями тестов выражения «*case tag*» являются имена объединения, указанного управляющим выражением. Значения тестов не должны повторяться.

```
type NodeType := union[tail; link: NodeType; data: integer]
… // значение node принадлежит типу NodeType
case tag node : union[tail; link: NodeType; data: integer]
  of link then Traverse(node.link)
  of data then node.data
  of tail then 0
end case
```

5.5. Циклические выражения

Форма циклического выражения такова: «заголовок цикла тело цикла эпилог цикла». Заголовок цикла может быть пустым или задаваться следующим образом: «*for* генератор диапазона ; определения имён начальных значений», «*for* генератор диапазона», «*for* определения имён начальных

значений», где генератор диапазона должен завершаться точкой с запятой, если после него идёт непустое тело цикла[54]. Тело цикла может быть пустым или задаваться следующим образом: «тест», «тест *do* определения имён циклических значений», «*do* определения имён циклических значений тест» или «*do* определения имён циклических значений». Если заголовок цикла не содержит генератор диапазона, то тело цикла должно содержать тест. Тест имеет вид «*while* булевское условие» или «*until* булевское условие». Эпилог цикла имеет вид «*returns* список редукций *end* первое ключевое слово циклического выражения[55]», где элементы списка редукций разделяются запятыми.

Например, в данном примере итеративно вычисляется число π:

```
for Approx := 1.0; Sign := 1.0; Denom := 1.0; i := 1
while i <= Cycles do
  Sign := - old Sign; Denom := old Denom + 2.0;
  Approx := old Approx + Sign / Denom;
  i := old i + 1
returns value of Approx * 4.0
end for
```

В следующем примере также итеративно вычисляется число π (с тем же результатом что и в предыдущем примере при чётном значении Cycles):

```
for i in 1..Cycles/2; do val := 1.0 / (4*i-3):real — 1.0 / (4*i-1):real
returns sum of val end for * 4.0
```

Циклическое выражение параллельно, если оно не содержит теста, его генератор диапазона не перечисляет потоки, оно не содержит редукций «*stream*», все функции начальных значений пользовательских редукций помечены прагмой «identity», все собирающие функции пользовательских редукций помечены прагмой «associative», оно не содержит «*old*» имён, определения имён циклических значений в правых частях не содержат имён, которые потом определяются в левой части. Циклическое выражение асинхронно параллельно, если все собирающие функции пользовательских редукций дополнительно помечены прагмой «commutative».

5.5.1. Заголовок и тело цикла

Циклическое выражение управляется тестом или генератором диапазона или тем и другим одновременно. Циклическое выражение, управляемое

[54] Для устранения неоднозначности разбора последнего триплета в генераторе диапазона.

[55] Первым ключевым словом циклического выражения могут быть ключевые слова «*for*», «*while*», «until» и «*do*».

тестом, выполняется пока «*when*» булевское условие истинно или «*until*» булевское условие ложно. Условие проверяется до или после тела цикла. Циклическое выражение, управляемое диапазоном, выполняется до тех пор, пока все элементы диапазона не будут перебраны. Если циклическое выражение управляется тестом и диапазоном одновременно, то оно выполняется до тех пор, пока позволяет тест и диапазон.

Генератор диапазона задаётся понятием «положение», введенным в разделе 4.3.1 для выражений конструирования массивов со следующими изменениями. Нет ограничений на количество элементов списка, задаваемого количеством размерностей массива. Элементы списка положения разделяются не запятой, а ключевым словом «*cross*». Элементами списка могут быть только обязательно именованные триплеты, потоки и массивы (не обязательно целого типа). Выражения не могут быть элементами списка. Неуказанные нижняя и верхняя границы триплетов генератора диапазона равны единице и бесконечности (положительной для положительного шага и отрицательной для отрицательного шага).

```
i in 1..3         // 1, 2, 3
j in ..3          // 1, 2, 3
k in 3..100..2    // 3, 5, …, 99
l in 100..98..-1 // 100, 99, 98
m in 1..          // 1, 2, 3, …
n in ..           // 1, 2, 3, …

// следующее выражение суммирует числа от 1 до N
for i in 1..N returns sum of i end for
// следующее выражение суммирует значения потока,
// удовлетворяющие контракту add1 из раздела 5.5.4
for x in S returns sum of x end for
// следующее выражение равно 24 (выполняется две итерации [1,3] и [2, 4])
for i in 1..2 dot j in 3..4 returns product of i+j end for
// следующее выражение равно 600 (итерации [1,3], [1, 4], [2, 3] и [2, 4])
for i in 1..2 cross j in 3..4 returns product of i+j end for
// следующее выражение равно 14400
for i in 1..2 cross j in i..4 returns product of i+j end for
```

Для массивов генератора диапазона допускается указание перечисляемых размерностей путём добавления суффикса «*at* [список имён индексов перечисляемых размерностей]». Список имён индексов перечисляемых размерностей должен иметь число элементов, равное размерности массива, и содержать, по крайней мере, одно имя индекса. Не перечисляемые размерности обозначаются двоеточием «..». Имя массива задаёт значение, полученное выражением выбора элементов массива с положением, равным списку имён индексов перечисляемых размерностей. Если суффикс «*at*»

отсутствует, то перечисляются все размерности, и имя массива имеет тип элемента массива. Массив генератора диапазона задаёт количество размерностей формы положения, равное числу своих перечисляемых размерностей.

```
// для трёхмерного массива «A» значение «x» является двухмерным массивом
// значение «i» пробегает все индексы первой размерности массива «A»
for x in A at i,..,..; do … x[..,..] … returns … end for
// в следующем примере перебираются все элементы массива «A», что
// эквивалентно заголовку цикла «for x in A at [i,j,k]»
for x in A; do … x … returns … end for
```

Определения имён начальных значений семантически полностью эквивалентны их заданию в окружающем выражении «*let*». Имена начальных значений и имена значений, определённых извне цикла, называются константами цикла. Определения имён циклических значений на каждой итерации цикла переопределяет значения указанных имён, так что на следующей итерации эти имена, использованные раньше в повторяемой части циклического выражения, будут иметь новые значения (определяемые имена должны иметь тип, эквивалентный или неявно приводимый к типу константы цикла с этим именем). После определения (текстуально, начиная с правой части этого определения до окончания циклического выражения) циклического имени можно обращаться к его значению на предыдущей итерации через «*old* имя», если существует константа цикла с этим именем.

```
// следующее выражение равно 91
for i := 1 while i < 5 do k := i; i := old i + 2; j := k + i
returns product of i+j end for
// следующее выражение цикла семантически эквивалентно предыдущему
// выражение «let» возвращает значения 91, 1
let i := 1 in while i < 5 do k := i; i := old i + 2; j := k + i
              returns product of i+j end while,
              I
end let
```

5.5.2. Эпилог цикла

Итерации цикла определяют редуцируемую последовательность значений для каждого имени, задаваемого генератором диапазона и телом цикла. Редукции формируют из редуцируемых последовательностей одно или несколько возвращаемых значений циклического выражения. Редукция имеет вид «*stream of* выражение фильтр», «*array of* выражение фильтр», «*array* форма свободного вида *of* выражение», «*array* форма фиксированного вида *of* выражение *at* список выражений положения» или «пользовательская редукция фильтр». Фильтр имеет вид «*when* булевское условие» или «*unless*

булевское условие» и включает циклические значение в редуцируемую последовательность только в случае, если «*when*» булевское условие истинно или «*unless*» булевское условие ложно.

Редукция «*stream*» формирует поток из своей редуцируемой последовательности. Редукция «*array*» без формы формирует одномерный массив с нижней границей равной единице из своей редуцируемой последовательности. Редукция «*array*» с формой свободного вида может использоваться только при отсутствии теста цикла и должна задавать форму с размерностью, равной размерности формы генератора диапазона. Редукция «*array*» с формой свободного вида конструирует массив с формой генератора диапазона, элементы которого задаются редуцируемой последовательностью и укладываются в конструируемый массив согласно форме генератора диапазона.

Границы формы фиксированного вида редукция «*array*» должны задаваться константами цикла. По-умолчанию, необязательная нижняя граница размерностей предполагается равной единице. Список выражений положения должен содержать число выражений, равное размерности конструируемого массива. Выражения должны быть целого типа и для каждой итерации цикла задавать различные элементы массива, иначе возвращается ошибочное значение массива. Все незаданные элементы массива (включая элементы с ошибочным или выходящим за границы массива положением) равняются ошибочным значениям.

```
// следующее выражение возвращает значение stream [2, 3, 4]
for i in 1..2 cross j in i..2 returns stream of i+j end for
// следующее выражение возвращает значение array of [2, 3, 4]
for i in 1..2 cross j in i..2 returns array of i+j end for
// следующее выражение возвращает значение array of [2, 3, 4]
for i in 1..2 cross j in i..2
returns array [1..3] of i+j at (i-1)*2 + j end for
// следующее выражение строит две одинаковые матрицы
let A := array [1..2, 0..1] of [1,.. := 1, 2; 2,.. := 3, 4];
    B := for i in 1..4
         returns array [1..2, 0..1] of i at i / 2 + 1, 1 - i % 2 end for
in A, B end let
// следующее выражение возвращает значение [3, 4, 6, 8]
for i in 1..2 cross j in 3..4 returns array of i*j end for
// выражение определяет массив с формой и границами массива «A»
for x in A at i, j; do k := … returns array [..,..] of g(x, k) end for
// следующее выражение возвращает значение array [4..7] of [40, 30, 20, 10]
for i in 1..4 returns array [4..7] of i * 10 at 4+i-1 end for
```

Пользовательские редукции рассматриваются в разделе 5.5.3. Предопределяются редукция «value», возвращающая последнее значение редуцируемой последовательности, редукция «sum», возвращающая сумму значе-

ний редуцируемой последовательности, редукция «product», возвращающая произведение значений редуцируемой последовательности, редукция «greatest», возвращающая наибольшее значение редуцируемой последовательности, редукция «least», возвращающая наименьшее значение редуцируемой последовательности. Также предопределяется редукция «catenate», возвращающая поток или массив, склеенный из потоков или массивов редуцируемой последовательности.

Если редуцируемая последовательность значений для редукции пуста (генератор цикла задаёт пустой диапазон значений, тест перед циклом был не удовлетворён или фильтр не пропустил ни одного значения), то редукция возвращает значения по умолчанию. Редукция «*stream*» возвращает пустой поток. Редукция «*array*» возвращает пустой массив для массивов, у которых не задана форма фиксированного вида, а для массивов с формой фиксированного вида возвращается ошибочное значение. Для пользовательских редукций по умолчанию возвращается начальное значение.

```
// данное выражение возвращает массив нечетных целых чисел
for i in 1..N returns array of i when i % 2 != 0 end for
```

5.5.3. Пользовательские редукции

Редукцией является объединение функций специального вида, участвующих в формировании результатов циклических выражений. Конструкция вызова редукции в предложении возврата циклического выражения выглядит следующим образом: «имя редукции (список значений начальных параметров редукции) *of* (список значений циклических параметров редукции)» или «имя редукции *of* (список значений циклических параметров редукции)», если у редукции нет начальных параметров. Если указан один циклический параметр редукции, то скобки вокруг него можно опускать. Значения начальных параметров редукции должны задаваться константами цикла. Имя редукции может обозначать значение типа запись с именами полей «ini», «rep», «join» и «ret», которые должны содержать значения функций, назначение которых объяснено ниже. Вместо имени редукции можно использовать конструктор указанного типа записи.

Если имя редукции (которое может предваряться именем модуля «имя модуля .») обозначает имя функции **A** (а не значения), то в качестве функции («ini»), формирующей начальное редукционное значение типа **T**, берётся функция с именем **A** и формой ближе всего[56] к типам значений начальных параметров редукции. Данная функция должна возвращать одно

[56] Алгоритм определения наиболее подходящей формы функции находится в разделе 4.6.

значение типа **T**. В качестве функции, формирующей начальное редукционное значение, может использоваться (если не подходит ни одна другая функция) обобщенная функция со свободными параметрами, число которых равно количеству циклических параметров редукции. Свободные параметры обобщенной функции задают типы соответствующих циклических параметров редукции. Функция «ini» может быть помечена прагмой «identity», если возвращаемое значение является единичным значением типа **T** относительно операции «join», описываемой ниже: «join(a, ini()) = join(ini(), a) = a».

В качестве функции («rep»), перевычисляющей редукционное значение, берется функция с именем **A** с формой, которая содержит тип **T** в качестве первого параметра и другими типами ближе всего к типам значений циклических параметров редукции. Данная функция также должна возвращать одно значение типа **T**.

Если объявлена собирающая функция («join») с именем **A**, которая берёт на входе два аргумента типа **T** и возвращает значение типа **T** (эта функция может совпадать с функцией «rep»), то она используется для сбора различных значений редукции, вычисленных параллельно. Если начальное редукционное значение было построено функцией, не помеченной прагмой «identity», то определять отдельную функцию «join» смысла нет. Объявление функции «join» можно помечать прагмой «associative», если функция ассоциативна: «join(join(a, b), c) = join(a, join(b, c))». Объявление функции «join» можно помечать прагмой «commitative», если функция коммутативна: «join(a, b) = join(b, a))». Если функция «join» ассоциативна, то она будет использована для параллельного вычисления значений редукции, тем самым определять отдельную не ассоциативную функцию «join» смысла нет. Если функция «join» ассоциативна и коммутативна, то она будет использована для (более эффективного) асинхронного параллельного вычисления значений редукции. Прагмы «identity», «associative» и «commutative» собираются вместе со всех объявлений одной функции.

Результирующие значения редукции определяются возвращающими значениями функции («ret») с именем **A** и формой формальных параметров, состоящей из одного типа **T**.

5.5.4. Предопределённые редукции

В интерфейсе модуля «std» определяются следующие функции, задающие предопределённые редукции:

```
// редукция, возвращающая последнее из редуцируемых значений
// или ошибочное значение по умолчанию
type value_red[T] := T
contract any[T] end contract
function value of any[T] (returns value_red[T])
function value of any[T] (value_red[T], T returns value_red[T])
function value of any[T] (value_red[T] returns T)

// редукция, возвращающая сумму редуцируемых значений
// или нулевое значение по умолчанию
type sum_red[T] := T
contract add1[T]
  operation + (T, T returns T)
  function zero (returns T)
end contract
//$ identity
function sum of add1[T] (returns sum_red[T])
function sum of add1[T] (sum_red[T], T returns sum_red[T])
/*$ commutative*/ /*$ associative*/
function sum of add1[T] (sum_red[T], sum_red[T] returns sum_red[T])
function sum of add1[T] (sum_red[T] returns T)
function zero (returns integer)
function zero (returns real)

// редукция, возвращающая перемноженные редуцируемые значения
// или единичное значение по умолчанию
type mul_red[T] := T
contract mul1[T]
  operation * (T, T returns T)
  function one (returns T)
end contract
//$ identity
function product of mul1[T] (returns mul_red[T])
function product of mul1[T] (mul_red[T], T returns mul_red[T])
/*$ commutative*/ /*$ associative*/
function product of mul1[T] (mul_red[T], mul_red[T] returns mul_red[T])
function product of mul1[T] (mul_red[T] returns T)
function one (returns integer)
function one (returns real)

// редукция, возвращающая наименьшее из редуцируемых значений
// или максимальное  значение по умолчанию
type min_red[T] := T
contract cmp1[T]
  operation < (T, T returns T)
  function min (returns T)
  function max (returns T)
end contract
//$ identity
function least of cmp1[T] (returns min_red[T])
function least of cmp1[T] (min_red[T], T returns min_red[T])
```

```
/*$ commutative*/ /*$ associative*/
function least of cmp1[T] (min_red[T], min_red[T] returns min_red[T])
function least of cmp1[T] (min_red[T] returns T)
function min (returns integer)
function min (returns real)
function max (returns integer)
function max (returns real)

// редукция, возвращающая наибольшее из редуцируемых значений
// или минимальное значение по умолчанию
type max_red[T] := T
//$ identity
function greatest of cmp1[T] (returns max_red[T])
function greatest of cmp1[T] (max_red[T], T returns max_red[T])
/*$ commutative*/ /*$ associative*/
function greatest of cmp1[T] (max_red[T], max_red[T] returns max_red[T])
function greatest of cmp1[T] (max_red[T] returns T)

// редукция, возвращающая конкатенацию массивов или потоков
// или пустой массив или поток по умолчанию
type cat_red[T] := T
contract cat1[T]
  operation || (T, T returns T)
  function empty (returns T)
end contract
//$ identity
function catenate of cat1[T] (returns cat_red[T])
function catenate of cat1[T] (cat_red[T], T returns cat_red[T])
//$ associative
function catenate of cat1[T] (cat_red[T], cat_red[T] returns cat_red[T])
function catenate of cat1[T] (cat_red[T] returns T)
function empty of any[T] (returns array of T)
function empty of any[T] (returns stream of T)
```

6. ИНТЕРФЕЙС ВЗАИМОДЕЙСТВИЯ С ДРУГИМИ ЯЗЫКАМИ

Из программ на языке Sisal можно получать доступ к функциям на языках Си++, Си и любых других языков программирования, которые поддерживают использование своего кода из языка Си или Си++[57]. Из других языков программирования, которые поддерживают вызов внешних функций на языке Си, возможно использование необобщенных процедур языка Sisal.

[57] Например, язык Фортран-2003 поддерживает средства взаимодействия с языком Си.

6.1. Доступ к функциям языка Sisal из других языков

Конечной целью трансляции модуля на языке Sisal является единица компиляции на языке Си++, поэтому для использования процедур языка Sisal, объявленных в интерфейсе модуля, из программы на языке Си++ в ней достаточно подключить заголовочный файл «sisal.hpp», описать прототипы внешних функций, придерживаясь правил, описанных в разделе 9, скомпилировать и слинковать всё вместе.

Транслятор языка Sisal для обеспечения поддержки языка Си генерирует функции-оболочки на этом языке для всех процедур, объявленных в интерфейсе модуля. Обобщенные процедуры вызывать из программ на языке Си не разрешается[58]. Поддержка вызова процедур языка Sisal из программ на языке Фортран осуществляется средствами языка Фортран-2003, обеспечивающими вызов функций языка Си из программ на языке Фортран.

Имена функций на языке Си++ и языке Си совпадают. Пользовательские типы задаются встроенными типами, лежащими в их основе. Все аргументы функций языка Sisal передаются по значению. Функции языка Sisal, возвращающие несколько значений, возвращают их с помощью записи языка Си, поля которой имеют типы, соответствующие типам языка Sisal на языке Си, которые возвращаются функцией в естественном порядке их следования. Ниже приведено описание простых встроенных типов языка Си из заголовочного файла «sisal.h», задающих типы языка Sisal:

```
// пустой тип
enum SisalNull { nil };

// булевский тип
enum SisalBooleanType { False, True };
struct SisalBoolean {
  SisalBooleanType error[59];
  SisalBooleanType value;
};

// символьный тип
struct SisalCharacter {
  SisalBooleanType error;
  SisalCharacterType value;
};

// целый тип
struct SisalInteger {
```

[58] Это ограничение происходит из того, что обобщённые процедуры задаются шаблонами языка Си++, которые невозможно использовать с помощью средств языка Си.

[59] Флаг ошибки, равный *true*, если значение типа ошибочно, и *false* иначе.

```
  SisalBooleanType error;
  SisalIntegerType value;
};

// вещественный тип
struct SisalReal {
  SisalBooleanType error;
  SisalRealType value;
};
```

Составной тип записи языка Sisal на языке Си задаётся как структура, у которой первое поле, задающее флаг ошибочности записи, имеет тип SisalBooleanType, а типы остальных полей задают соответствующие типы полей записи языка Sisal на языке Си.

Составной тип объединения языка Sisal на языке Си задаётся как структура с двумя полями. Первое поле имеет тип *int* и содержит номер текущего тега объединения или отрицательное значение, если значение объединения ошибочно. Нумерация тегов ведётся с нуля, а сами теги упорядочены естественным порядком их задания в объединении языка Sisal. Второе поле содержит объединение с типами тегов объединения языка Sisal на языке Си в их естественном порядке следования.

Составной тип массива языка Sisal на языке Си задаётся как структура с четырьмя полями. Первое поле типа *int* содержит количество размерностей массива Dim (отсчитываемое с единицы) или значение меньше или равное нулю, если значение массива ошибочно. Второе поле с типом указателя на значение типа SisalIntegerType указывает на первый элемент массива с размерностью Dim, содержащего значения нижних границ размерностей, начиная с внешней размерности. Третье поле с типом указателя на значение типа SisalIntegerType указывает на первый элемент массива с размерностью Dim, содержащего значения верхних границ размерностей, начиная с внешней размерности. Четвёртое поле с типом указателя на элемент типа, соответствующего типу элемента массива языка Sisal на языке Си, указывает на первый элемент массива с размерностью, равной произведению количества элементов каждой размерности[60]. Элементы многомерного массива располагаются в row-major порядке. Памятью указателей входных (в функцию языка Sisal на языке Си) массивов нужно управлять самостоятельно. В обязанности вызывающего Си-кода входит освобождение памяти (функцией free), занимаемой указателями возвращаемых массивов.

[60] Количество элементов размерности массива равняется верхней границе размерности минус нижняя граница размерности плюс один.

Составной тип потока языка Sisal на языке Си задаётся как структура с шестью полями. Первое поле с типом указателя «*void**» указывает на некоторое состояние потока или задаёт ошибочное значение потока, если указатель равен нулю. Второе поле является указателем на функцию empty языка Sisal. Третье поле является указателем на функцию first языка Sisal. Четвёртое поле является указателем на функцию rest языка Sisal. Пятое поле является указателем на функцию, которая берёт и возвращает указатель «*void**», копируя состояние потока. Шестое поле является указателем на функцию, которая берёт указатель «*void**» и освобождает состояние потока им задаваемым. Памятью состояний входных потоков нужно управлять самостоятельно. В обязанности вызывающего Си-кода входит освобождение памяти, занимаемой состоянием возвращаемых потоков.

Тип функции языка Sisal на языке Си задаётся указателем на функцию, удовлетворяющую правилам описания функций языка Sisal на языке Си.

6.2. Интерфейс модуля на другом языке

Модули, реализованные на других языках программирования, называются инородными модулями. В интерфейсе инородного модуля указывается имя языка его реализации. Поддерживаются языки Си («C»), Си++ («Cpp») и Фортран («Fortran»). Содержимое интерфейса инородного модуля имеет специальный синтаксис.

Интерфейс инородного модуля может содержать объявления инородных процедур, определения переименованных и пользовательских типов и конструкции импорта переименованных и пользовательских типов других модулей.

Объявление инородной функции выглядит как «*function* имя функции (список формальных параметров *returns* тип возвращаемого значения)», где некоторые типы формальных параметров может предваряться ключевым словом «*in*», «*out*», «*in out*» или «*raw*». Ключевое слово «*out*» или «*raw*» может предварять тип возвращаемого значения. Использование любого из перечисленных ключевых слов возможно только перед множеством типов **S**, которому принадлежат инородные типы, пользовательские типы, в основании которых лежит тип из **S**, массивы от типа из **S**, записи с типами полей из **S** и объединения с типами тегов из **S** (рекурсивные зависимости объединений не допускаются). Далее при рассмотрении применимости этих ключевых слов, без дополнительных упоминаний, пользовательский тип рассматривается как встроенный тип, на котором он основан.

Ключевое слово «*raw*» может использоваться только перед типами записей и объединений. Ключевое слово «*raw*» перед типом записи означает, что в инородную процедуру передается (или для ключевого слова «*raw*» перед возвращаемым типом из инородной процедуры возвращается) структура с типами полей, соответствующих типам полей записи. Полям с типом записи или объединения, соответствуют поля типа «*raw*» запись и объединение. Полям типа массив со свободной формой, соответствует указатель в динамической памяти на последовательность всех его элементов (смотри описание способа передачи «*in* массива» ниже). Полям типа массив с фиксированной формой соответствует последовательность всех его элементов в записи. Ключевое слово «*raw*» перед типом объединения означает, что в инородную процедуру передается (или для ключевого слова «*raw*» перед возвращаемым типом из инородной процедуры возвращается) союз с типами полей, соответствующих типам тегов объединения таким же образом, как и в «*raw*» записи.

Ключевое слово «*in*» может использоваться перед всеми типами из **S**. Ключевое слово «*in*» означает, что в инородную процедуру передаётся указатель на некоторую выделенную динамическую память, содержащую копию значение типа. После вызова инородной процедуры копия значения типа и динамическая память, которую она занимает, освобождаются. Ключевое слово «*in*» перед массивом означает, что он задаётся указателем на последовательность всех его элементов. Элемент массива, являющийся массивом, задаётся указателем на выделенную динамическую память, содержащую последовательность всех его элементов. Элементы массивы, являющиеся записью или объединением, задаются непосредственно как «*raw*» записи или объединения, соответственно. Ключевое слово «*in*» перед записью или объединением означает, что они задаются указателем на «*raw*» записи или объединения, соответственно.

Ключевые слова «*in out*» могут использоваться перед типами множества $\mathbf{S_2}$, которому принадлежат инородные типы, массивы от типа из $\mathbf{S_2}$ и записи с типами полей из $\mathbf{S_2}$. Использование ключевых слов «*in out*» означает, что, как и для ключевого слова «*in*», в инородную процедуру передаётся указатель на динамическую память, только после вызова процедуры из копий значений в динамической памяти формируется возвращаемое значение типа формального параметра, которое считается дополнительным возвращаемым значением инородной процедуры. Размерности форм массивов считаются неизменными.

Ключевое слово «*out*» может использоваться перед типами из множества **S3**, которому принадлежат инородные типы, массивы с фиксированной

формой от типа из **S3**, записи с типами полей из **S3**. Для формального параметра с ключевым словом «*out*» в инородную процедуру передаётся (или для ключевого слова «*out*» перед возвращаемым типом из инородной процедуры возвращается) указатель на неинициализированный объект инородного типа или указатель на неинициализированную динамическую память с размером достаточным для хранения элементов массива с фиксированной формой или записи. После завершения работы инородной процедуры неинициализированная память считается содержащей возвращаемое значение, которое считается дополнительным возвращаемым значением инородной процедуры. Так же входное значение формального параметра с ключевым словом «*out*» не указывается, как если бы его не было.

Объявление инородной операции выглядит как «*operation* знак операции *is* имя функции (список формальных параметров *returns* тип возвращаемого значения)» или «*operation* знак операции *is* имя функции (список формальных параметров)», если список формальных параметров содержит хотя бы одно ключевое слово «*out*». Число и типы формальных параметров и возвращаемых значений операции должны удовлетворять требованиям, приведённым в табл. 1 с учётом действий ключевых слов «*in*» и «*out*».

В объявлении инородной процедуры можно не указывать тип возвращаемого значения вместе с ключевым словом «*returns*», если список формальных параметров содержит хотя бы одно ключевое слово «*out*»[61]. После объявления инородной процедуры можно указывать имя языка «*in* имя языка», если оно отличается от имени языка задаваемого интерфейсом инородного модуля, в котором оно содержится[62].

Объявление инородной процедуры может содержать прагму «weight = целочисленное выражение», где указанное целочисленное выражение задаёт значение веса, обозначающее приблизительное количество тактов, необходимых в среднем для исполнения данной процедуры. Для корректного сравнения весов инородных процедур различных модулей, полученных в разное время для разных архитектур, на уровне проекта программы можно задавать множитель, на который множатся все веса инородных процедур в указанном интерфейсе модуля. Прагма «weight» переопределяет значение веса в переобъявлении процедуры.

В интерфейсе инородного модуля на языке Си++ нельзя объявлять две процедуры с одинаковым именем (для инородной операции рассматривается её имя функции), одинаковой формой и разной формой возвращаемых

[61] Это требование позволяет сохранить хотя бы одно возвращаемое значение процедуры для языка Sisal.

[62] Например, в модуль на языке Си++ может экспортировать функции на языке Си++ и Си.

значений. Форма инородной процедуры включает признак передачи по указателю для каждого формального параметра, перед типом которого используются ключевые слова «*in*», «*out*» и «*in out*», и признак непосредственной передачи для каждого формального параметра, перед типом которого используются ключевое слово «*raw*». Формы инородных функций совпадают, если дополнительно совпадают указанные признаки типов формальных параметров.

В интерфейсе модуля на языке Си и Фортран не может быть объявлено более одной функции (включая имена функций операций) с одинаковым именем и разной формой. Модуль не может включать интерфейсы на языке Си и Фортран, которые содержат функции с одним именем и разной формой.

6.3. Предопределённые инородные типы

С компилятором языка Sisal для архитектуры x86 поставляется модуль с именем «Cpp», реализованный на языке Си++, который имеет следующий интерфейс на языке Sisal:

```
interface Cpp
  // тип указателя для возвращаемых массивов в динамической памяти (malloc)
  type ptr := "void*"
  contract any[T] end contract // на тип T нет никаких ограничений
  // конструирует массив из указателя, нижней и верхней границы массива,
  // освобождая (функцией free) память указателя;
  // если тип T не инородной тип или пользовательский тип,
  // в основе которого лежит инородной тип,
  // то возвращается ошибочное значение массива
  function make_array of any[T](ptr, integer, integer returns array of T)
  // запрещает копирование и запрещает объявлять и определять эту операцию
  no operation (ptr returns ptr)
  operation (null returns ptr) // error[ptr] это нулевой указатель

  // тип указателя для возвращаемых массивов в статической памяти
  type ptr_s := "void*"
  // конструирует массив из указателя;
  // если тип T не инородной тип или пользовательский тип,
  // в основе которого лежит инородной тип,
  // то возвращается ошибочное значение массива
  function make_array of any[T](ptr_s, integer, integer returns array of T)
  // запрещает копирование и запрещает объявлять и определять эту операцию
  no operation (ptr_s returns ptr_s)
  operation (null returns ptr_s) // error[ptr_s] это нулевой указатель

  // строки в динамической памяти, оканчивающиеся нулевым символом
  type psz := "char*"
  operation (array of character returns psz)
```

```
operation (psz returns array of character)
operation (psz returns psz)  // копирование строк возможно
operation (psz returns null)
operation (null returns psz) // error[psz] это нулевой указатель

// строки в статической памяти, оканчивающиеся нулевым символом
type psz_s := "char*"
operation (psz returns psz_s)
operation (psz_s returns psz)
operation (psz_s returns psz_s) // копируется указатель
operation (null returns psz) // error[psz_s] это нулевой указатель

// широкие строки в динамической памяти, оканчивающиеся нулевым символом
type pwsz := "wchar_t*"
operation (array of character returns pwsz)
operation (pwsz returns array of character)
operation (pwsz returns pwsz) // копирование широких строк возможно
operation (pwsz returns null)
operation (null returns pwsz) // error[pwsz] это нулевой указатель

// широкие строки в статической памяти, оканчивающиеся нулевым символом
type pwsz_s := "wchar_t*"
operation (pwsz returns pwsz_s)
operation (pwsz_s returns psz)
operation (pwsz_s returns pwsz_s) // копируется указатель
operation (null returns pwsz_s) // error[pwsz_s] это нулевой указатель

// булевский тип языка Си++
type bool = "bool"
operation (boolean returns bool)
operation (bool returns boolean)

// символьные типы языка Си и Си++
type char = "char"
type wchar = "wchar_t"
operation (character returns char)
operation (character returns wchar)
operation (char returns character)
operation (wchar returns character)

// архитектурно-зависимые типы целых чисел
type int8 = "__int8"
type int16 = "__int16"
type int32 = "__int32"
type int64 = "__int64"
operation (integer returns int8)
operation (integer returns int16)
operation (integer returns int32)
operation (integer returns int64)
operation (int8 returns integer)
operation (int16 returns integer)
```

```
  operation (int32 returns integer)
  operation (int64 returns integer)

  // архитектурно-зависимые типы вещественных чисел
  type real32 = "float"
  type real64 = "double"
  type real80 = "long double"
  operation (real returns real32)
  operation (real returns real64)
  operation (real returns real80)
  operation (real32 returns real)
  operation (real64 returns real)
  operation (real80 returns real)

  …  // другие операции инородных типов
end interface
```

7. СИНТАКСИЧЕСКАЯ И ЛЕКСИЧЕСКАЯ СТРУКТУРА ЯЗЫКА

В этом разделе приводится полное описание синтаксической и лексической структуры языка Sisal 3.2. Описание приводится в терминах грамматики ANTLR v3 [14] и лежит в классе LL_4 [14] языков.

Грамматика ANTLR содержит правила вида «нетерминал : альтернативы ;». Альтернативы разделяются знаком «|». Альтернатива состоит из (возможно пустой) последовательности термов, разделённых пробелами. Термом может быть нетерминал, строка терминалов, заключённых в одинарные кавычки, и альтернативы, заключённой в скобки. Терм, оканчивающийся знаком «?», является необязательным. Терм, оканчивающийся знаком «+», может повторяться один или более раз. Терм, оканчивающийся знаком «*», может повторяться ноль или более раз. Комментарии задаются как в языке Си++.

При описании лексической структуры текста используются дополнительные обозначения. Ключевое слово «fragment» перед правилом означает, что оно задаёт часть лексемы, используемую в другом правиле. Последовательности терминальных символов могут включать коды обратной косой черты. Можно задавать отрезки терминальных символов в виде «символ нижней границы отрезка .. символ верхней границы отрезка». Терм, задающий терминальные символы и предварённый знаком «~», обозначает все символы, кроме указанных.

7.1. Утверждения языка

Ниже приводится общая структура единицы компиляции программы:

```
compilation_unit:
  'module' mod_id (def_stmt ';'?)* 'end' 'module' |
  'interface' mod_id (decl_stmt ';'?)* 'end' 'interface' |
  'interface' mod_id 'in' lang_id (out_decl_stmt ';'?)* 'end' 'interface';

def_stmt:  def_import_stmt | type_def_decl | contract_def |
          'forward' func_decl | 'forward' op_decl | func_def | op_def;

decl_stmt: decl_import_stmt | type_def_decl | contract_def |
           func_decl | op_decl;

out_decl_stmt: type_def | out_func_decl | out_op_decl;
```

Далее указано устройства конструкций импорта:

```
decl_import_stmt: 'import' mod_id (',' mod_id)* | 'import' mod_id
                  (':' | '-') decl_import_obj (',' decl_import_obj)*;

decl_import_obj:  LexName | 'type' type_id | 'contract' contract_id;

def_import_stmt:  'import' mod_id (',' mod_id)* | 'import' mod_id
                  (':' | '-') def_import_obj (',' def_import_obj)*;

def_import_obj:
  LexName |
  'type' type_id |
  'contract' contract_id |
  'function' func_id '[' '..' ']' |
  'function' func_id of_param_names?
    ('[' out_types? ('returns' types)? ']') |
  'operation' op_cast of_param_names? '[' type 'returns' type ']' |
  'operation' op_1 of_param_names? '[' type ']' |
  'operation' op_12 of_param_names? '(' type (',' type)? ']' |
  'operation' op_2 of_param_names? '[' type ',' type ']' |
  'operation' op_func of_param_names? '[' types ']';

of_param_names: 'of' param_names;

param_names: '[' param_id (',' param_id)* ']';

op_all:  op_cast | op_1 | op_12 | op_2 | op_func;
op_cast: ':' |;
op_1:    '!' | '.' operation_id;
op_12:   '+' | '-';
op_2:    '*' | '/' | '%' | '**' | '^' | '&' |
         '|' | '||' | '[' ']' | '=' | '<';
op_func: '(' ')';
```

Описание структуры типов приводится ниже:

```
type_def_decl: type_def | type_decl;

type_def:  'type' type_id '=' type | 'type' type_id ':=' type;

type_decl: 'type' type_id param_names ':=' type;

type:  basic_type | foreign_type | type_ref | expr_type |
       array_type | stream_type | record_type | union_type | func_type;

types: type (',' type)*;

type_ref:    (mod_id '.')? type_id;
basic_type:  'null' | 'boolean' | 'character' | 'integer' | 'real';
foreign_type: LexStrConst;
expr_type:    'type' '(' expr ')';

array_type:     'array' (free_dim_form | fixed_dim_form) 'of' type;
free_dim_form:  '[' '..' (',' '..')* ']';
fixed_dim_form: '[' dim_duplet (',' dim_duplet)* ']';
dim_duplet:     expr? '..' expr;

stream_type: 'stream' 'of' type;

record_type: 'record' '[' field_spec (';' field_spec)* ';'? ']';
field_spec:  field_id (',' field_id)* ':' type;

union_type:  'union' '[' tag_spec (';' tag_spec)* ';'? ']';
tag_spec:    tag_id (',' tag_id)* (':' type | );

func_type:   'function' '(' types? 'returns' types ')';
```

Контракты определяются следующим образом:

```
contract_def: 'contract' contract_id param_names of_contract?
              (func_decl | op_decl)* 'end' 'contract';

of_contract:  'of' contract_id param_names;
```

Функции и операции объявляются и определяются следующим образом:

```
func_decl: 'function' func_id of_contract?
           '(' arg_decls? 'returns' types ')';

op_decl: 'no'? 'operation' (
      (op_cast | op_1) of_contract? '(' arg_decl 'returns' type ')' |
      op_12 of_contract? '(' arg_decl (',' arg_decl)? 'returns' type ')' |
      op_2 of_contract? '(' arg_decl ',' arg_decl 'returns' type ')' |
      op_func of_contract? '(' arg_decls 'returns' types ')'
    );

arg_decls: arg_decl (',' arg_decl)*;
arg_decl:  arg_def | type;
```

```
arg_defs:  arg_def (',' arg_def)*;
arg_def:   value_id ':' type;

func_def: 'function' func_id of_contract? '(' arg_defs? 'returns' types ')'
          exprs 'end' 'function';

op_def:   'operation' (
  (op_cast | op_1) of_contract? '(' arg_def 'returns' type ')' |
  op_12 of_contract? '(' arg_def (',' arg_def)? 'returns' type ')' |
  op_2 of_contract? '(' arg_def ',' arg_def 'returns' type ')' |
  op_func of_contract? '(' arg_defs 'returns' types ')'
)  exprs 'end' 'operation';
```

Иноязычные функции и операции объявляются, как указано ниже:

```
out_func_decl: 'function' func_id
               '(' out_args? ('returns' out_ret_types)? ')' ('in' lang_id)?;

out_op_decl:   'operation' op_all 'is' func_id
               '(' out_args ('returns' out_ret_types)? ')' ('in' lang_id)?;

out_args: out_arg (',' out_arg)*;
out_arg:  value_id ':' out_type | out_type;

out_types: out_type (',' out_type)*;
out_type:  ('raw' | 'in' | 'out' | 'in' 'out')? type;

out_ret_types: out_ret_type (',' out_ret_type)*;
out_ret_type:  ('raw' | 'out') type;
```

7.2. Выражения языка

Общий вид выражения приведён ниже (бинарные операции разбираются с учётом их приоритетов):

```
exprs: expr (',' expr)*;
expr:  concatenation;

concatenation:  disjunction ('||' disjunction)*;
disjunction:    xor ('|' xor)*;
xor:            conjunction ('^' conjunction)*;
conjunction:    equivalence ('&' equivalence)*;
equivalence:    relation ( ('=' | '!=') relation )*;
relation:       add_sub ( ('<' | '<=' | '>' | '>=') add_sub )*;
add_sub:        mul_div ( ('+' | '-') mul_div )*;
mul_div:       exponentiation ( ('*' | '/' | '%') exponentiation )*;
exponentiation: prefix_op ('**' exponentiation)?;

prefix_op:  ('+' | '-' | '!') prefix_op | postfix_op;

postfix_op: primary (
```

```
  '(' expr? (',' expr?)* ')' |
  '[' placement (':=' exprs (';' placed_elements)*)? ';'? ']' |
  'replace' '[' field_defs ']' |
  '.' LexName |
  'is' 'tag' tag_id |
  'is' 'error' |
  ':' safe_type |
  ':' '[' type ']'
)*;

safe_type: safe_type_ref | safe_array_type | safe_stream_type |
           basic_type | foreign_type | record_type | union_type |
           func_type | expr_type;

safe_type_ref:    mod_id '.' type_id;
safe_array_type:  'array' (free_dim_form | fixed_dim_form) 'of' safe_type;
safe_stream_type: 'stream' 'of' safe_type;
```

Синтаксис операндов выражения приведён ниже:

```
primary:  '(' expr ')' | 'old'? LexName | constant |
          union_gen | record_gen | stream_gen | array_gen |
          let_expr | if_expr | case_expr | for_expr;

constant: 'false' | 'true' | 'nil' | LexIntConst |
          LexRealConst | LexCharConst | LexStrConst |
          func_value | 'error' '[' type ']';

func_value:
  func_id '.' '[' types ']' |
  'function' func_id '[' '..' ']' |
  'function' func_id '[' out_types? ('returns' types)? ']' |
  'function' func_id '.' '[' types ']' '[' out_types? ']' |
  'function' func_id of_typed_params
    '[' out_types? ('returns' types)? ']' |
  'operation' op_cast of_typed_params? '[' type 'returns' type ']' |
  'operation' op_1 of_typed_params? '[' type ']' |
  'operation' op_12 of_typed_params? '(' type (',' type)? ']' |
  'operation' op_2 of_typed_params? '[' type ',' type ']' |
  'operation' op_func of_typed_params? '[' types ']' |
  'function' func_id? '(' arg_defs? 'returns' types ')'
    exprs 'end' 'function';

of_typed_params: 'of' '[' typed_param (',' typed_param)* ']';

typed_param: param_id '=' type;
```

Выражения конструирования объединений и записей указаны ниже:

```
union_gen:  'union' type '[' tag_id (':=' expr)? ']';
record_gen: 'record' type '[' (field_defs | ':=' exprs) ']' |
            'record' '[' field_defs ']';

field_defs: field_def (';' field_def)* ';'?;
field_def:  deep_field_name (',' deep_field_name)* ':=' exprs;
deep_field_name: field_id ('.' field_id)*;
```

Потоки конструируются, как указано ниже:

```
stream_gen: 'stream' 'of' type? '[' triplets ']';
triplets:  (triplet (',' triplet)*)?;
triplet:   expr triplet23? | triplet23;
triplet23: '..' expr? ('..' expr)?;
```

Синтаксис конструктора массива приведён ниже:

```
array_gen : ('array' 'of' type?)? '[' triplets ']' |
            'array' type '[' elements ']' |
            'array' named_fixed_dims ('of' type? | type) '[' elements ']' |
            'array' fixed_dim_names type '[' elements ']';

named_fixed_dims: '[' named_dim_duplet (',' named_dim_duplet)* ']';
named_dim_duplet: (dim_id 'in')? expr? '..' expr;

fixed_dim_names: '[' fixed_dim_name (',' fixed_dim_name)* ']';
fixed_dim_name:  dim_id | '..';

elements: ':=' exprs |
          placed_elements (';' placed_elements)*
          (';' ('else' ':=' exprs ';'?)?)?;
placed_elements: placement ':=' exprs;
placement:       placement_part (',' placement_part)*;
placement_part:  named_triplet ('dot' named_triplet)*;
named_triplet:   (dim_id 'in')? triplet;
```

Выражение «*let*» задаётся следующим образом:

```
let_expr:   'let' name_defs 'in' exprs 'end' 'let';
name_defs:  name_def (';' name_def)* ';'?;
name_def:   name_decls ':=' exprs;
name_decls: name_decl (',' name_decl)*;
name_decl:  value_id (':' type)?;
```

Синтаксис условных выражений приведён ниже:

```
if_expr: 'if' expr 'then' exprs ('elseif' expr 'then' exprs)*
         ('else' exprs)? 'end' 'if';

case_expr:     'case' expr ('of' pattern (',' pattern)* 'then' exprs)+
               ('else' exprs)? 'end' 'case' | case_tag_expr;

case_tag_expr: 'case' 'tag' expr ('of' tag_id (',' tag_id)* 'then' exprs)+
               ('else' exprs)? 'end' 'case';

pattern: expr ('..' expr?)? | '..' expr;
```

Циклические выражения описаны далее:

```
for_expr:    for_body for_test for_returns 'end' 'do' |
             while_test for_body for_returns 'end' 'while' |
             until_test for_body for_returns 'end' 'until' |
             'for' for_part for_returns 'end' 'for';
```

```
for_part:     for_test for_body |
              for_body for_test |
              range_gen (
                ';' name_defs (for_body for_test? | for_test for_body)? |
                ';' (for_body for_test? | for_test for_body) | ';'
              )?;

for_body:     'do' name_defs;

for_test:     while_test | until_test;

while_test:   'while' expr;

until_test:   'until' expr;

range_gen:    dot_gen ('cross' dot_gen)*;

dot_gen:      dim_gen ('dot' dim_gen)*;

dim_gen:      dim_id 'in' (expr ('at' dim_names | triplet23)? | triplet23);

dim_names:    dim_name (',' dim_name)*;

dim_name:     dim_id | '..';

for_returns: 'returns' reduction (';' reduction)* ';'?;

reduction:    ('stream' | 'array') 'of' expr filter? |
              'array' free_dim_form 'of' expr |
              'array' fixed_dim_form 'of' expr 'at' dim_names |
              (value_id | record_gen) ('(' exprs ')')? 'of' exprs filter?;

filter:       ('when' | 'unless') expr;
```

Далее приводится определение синонимов идентификатора:

```
mod_id:       LexName;
dim_id:       LexName;
param_id:     LexName;
operation_id: LexName;
contract_id:  LexName;
program_id:   LexName;
lang_id:      LexName;
func_id:      LexName;
type_id:      LexName;
tag_id:       LexName;
field_id:     LexName;
value_id:     LexName;
```

7.3. Лексическая структура языка

Текст модуля задан символами уникода (Unicode-16) [15] в UTF-8 [16] кодировке. Множество символов алфавита языка ограничено прописными и

строчными буквами латинского алфавита, арабскими цифрами и специальными символами, приведенными в таблице 4 вместе с их десятичными ASCII [12] кодами. Следующие символы называются пробельными и разделяют другие конструкции языка: табуляция (десятичный ASCII код 9), перевод строки (код 10), вертикальная табуляция (код 11), новая страница (код 12), возврат каретки (код 13) и пробел (код 32). Остальные символы, не принадлежащие алфавиту языка, могут входить только в состав комментариев, символьных и строковых литералов.

Таблица 4

Специальные символы

Знак	!	"	#	%	&	'	(	)	*	+	,	-	.	/
Код	33	34	35	37	38	39	40	41	42	43	44	45	46	47
Знак	:	;	<	=	>	@	[	\	]	^	_	{	|	}
Код	58	59	60	61	62	64	91	92	93	94	95	123	124	125

Ниже приводится формальное описание пробельных и нераспознаваемых символов:

```
// Whitespace is ignored
LexWhiteSpace: LexLineWhiteSpace | '\n';
fragment LexLineWhiteSpace: '\t' | '\000B' /*\v*/ | '\f' | '\r' | ' ';

// Unexpected character is ignored with warning.
LexUnknownChar: '\u0000'..'\u0008' | '\u000e'..'\u001f' |
                '#' | '$' | '?' | '@' | '\\' | '`' | '{' | '}' | '~' |
                '\u007F'..'\uFFFE';
```

Допустимы строковые комментарии, начинающиеся символами «//», и не вложенные друг в друга блочные (многострочные) комментарии, начинающиеся символами «/*» и оканчивающиеся символами «*/» (комментарий без завершения продолжается до конца файла). Строчный комментарий эквивалентен символу перевода строки, а блочный комментарий рассматривается как пробельный символ при трансляции. Комментарий, который начинается с символа доллара «$», называется прагмой и задаёт свойства конструкции, идущей следом (одной конструкции можно сопоставлять несколько прагм). Прагма может иметь вид «имя» или «имя = унарное выражение», где в унарном выражении могут принимать участие имена, видимые в месте расположения прагмы. Нераспознанные прагмы вызывают предупреждения компилятора. Ниже приводится описание комментариев и прагм:

```
LexLineCommentOrPragma: '/' '/' (
  // Pragma will be parsed later separately as "LexName ('=' expr)?".
  '$' ~'\n'* '\n'? |
  ~('$' | '\n') ~'\n'* '\n'? | '\n' // Comment is ignored.
);

LexBlockCommentOrPragma: '/' '*' (
  // Pragma will be parsed later separately as "LexName ('=' expr)?".
  '$' (options {greedy=false;}: .)* '*' '/' |
  // Comment is ignored.
  ~('$' | '*') (options {greedy=false;}: .)* '*' '/' |
  '*' ('/' | (options {greedy=false;}: .)* '*' '/')
);
```

Идентификаторы задаются цепочкой букв верхнего регистра и букв нижнего регистра (отличных от букв верхнего регистра), десятичных цифр и знака подчеркивания. Идентификатор не может начинаться с десятичной цифры и состоять из единственного знака подчеркивания. Далее идёт описание лексем идентификаторов и числовых литералов:

```
LexName:      LexLetter (LexLetter | LexDigit | '_')*;
LexDummyName: '_';
fragment LexLetter: 'A'..'Z' | 'a'..'z';
fragment LexDigit:  '0'..'9';

LexIntConst:  (LexDigit+ '#')? LexDigit+;

LexRealConst: LexDigit+ (LexRealExpField | '.' LexDigit* LexRealExpField?);
fragment LexRealExpField: ('E' | 'e') ('-' | '+')? LexIntConst;
```

Ниже приводится определение лексем символьных и строковых литералов:

```
LexCharConst: '\'' (
  (
    ( '\\' ( LexEscChar | LexEscCode |
       // Unrecognized escape-sequence.
       // Space character assumed.
       ~(LexEscChar | LexDigit | 'o' | 'h') |
       'o' ~LexOctDigit | 'h' ~LexHexDigit
         )
    ) | ~('\\' | '\'' | '\n')
  ) (
    '\'' |
    // A character constant can not contain not a single character.
    // Ignore extra characters until single quote or end of file.
    ~'\'' ~'\''* '\''?
  ) | (
    // A character constant can not be empty.
    // Space character assumed.
    '\'' |
```

```
    // A character constant may not extend across a line boundary.
    // Space character assumed.
    // Skipping until next single quote or end of file.
    '\n' ~'\''* '\''? |
    // A character constant may not contain end of file.
    // Space character assumed.
    EOF
  )
);

LexStrConst: (LexStr | LexRawStr)+;

fragment LexStr: '"' (
  // Semicolon after LexEscCode is ignored
  ( '\\' ( '"' | LexEscChar | LexEscCode (';' | ~('\\' | '"' | '\n' | ';')) |
     // Unrecognized escape-sequence.
     // Space character assumed.
     ~(LexEscChar | LexDigit | 'o' | 'h' | '"') |
     'o' ~LexOctDigit | 'h' ~LexHexDigit
               )
    ) | ~('\\' | '"' | '\n')
)* (
  '"' |
  // A character string constant may not extend across a line boundary.
  // End of string assumed.
  // Skipping until next double quote or end of file.
  '\n' ~'"'* '"'? |
  // A character string constant may not contain end of file.
  // End of string assumed.
  EOF
);

fragment LexRawStr: '@' '"' ( '\\' '"' | ~('\\' | '"' | '\n') )* (
  '"' |
  // A character string constant may not extend across a line boundary.
  // End of string assumed.
  // Skipping until next double quote or end of file.
  '\n' ~'"'* '"'? |
  // A character string constant may not contain end of file.
  // End of string assumed.
  EOF
);

fragment LexOctDigit: '0'..'7';
fragment LexHexDigit: '0'..'9' | 'A' .. 'F' | 'a' .. 'f';
fragment LexEscChar: '\'' | '\\' | 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v';
fragment LexEscCode: LexDigit+ | 'o' LexOctDigit+ | 'h' LexHexDigit+;
```

Ниже приводятся описание всех лексем, не определяемых правилами грамматики, а также список ключевых слов (42 штуки), которые не могут быть идентификаторами:

```
tokens {
  LexAnd   = '&';  LexAssign    = ':='; LexColon = ':';
```

```
  LexComma = ',';  LexConcat    = '||'; LexDiv   = '/';
  LexDot   = '.';  LexDots      = '..'; LexEQ    = '=';
  LexExp   = '**'; LexGE        = '>='; LexGT    = '>';
  LexLE    = '<='; LexLPar      = '(';  LexLSPar = '[';
  LexLT    = '<';  LexMinus     = '-';  LexMod   = '%';
  LexMul   = '*';  LexNE        = '!='; LexNOT   = '!';
  LexOr    = '|';  LexPlus      = '+';  LexRPar  = ')';
  LexRSPar = ']';  LexSemicolon = ';';  LexXOR   = '^';

  Lex_array     = 'array';     Lex_at       = 'at';
  Lex_case      = 'case';      Lex_contract = 'contract';
  Lex_cross     = 'cross';     Lex_do       = 'do';
  Lex_dot       = 'dot';       Lex_else     = 'else';
  Lex_elseif    = 'elseif';    Lex_end      = 'end';
  Lex_error     = 'error';     Lex_false    = 'false';
  Lex_for       = 'for';       Lex_forward  = 'forward';
  Lex_function  = 'function';  Lex_if       = 'if';
  Lex_import    = 'import';    Lex_in       = 'in';
  Lex_interface = 'interface'; Lex_is       = 'is';
  Lex_let       = 'let';       Lex_module   = 'module';
  Lex_nil       = 'nil';       Lex_no       = 'no';
  Lex_of        = 'of';        Lex_old      = 'old';
  Lex_operation = 'operation'; Lex_out      = 'out';
  Lex_raw       = 'raw';       Lex_record   = 'record';
  Lex_replace   = 'replace';   Lex_returns  = 'returns';
  Lex_stream    = 'stream';    Lex_tag      = 'tag';
  Lex_then      = 'then';      Lex_true     = 'true';
  Lex_type      = 'type';      Lex_union    = 'union';
  Lex_unless    = 'unless';    Lex_until    = 'until';
  Lex_when      = 'when';      Lex_while    = 'while';
}
```

7.4. Препроцессор

Транслятор языка Sisal 3.2 включает препроцессор, осуществляющий условную трансляцию, генерацию пользовательских предупреждений и ошибок, управление нумерацией строк и выделение именованных областей программы. В основе препроцессора лежит препроцессор языка C# [17]. Препроцессор языка является частью лексического анализатора и называется препроцессором для сохранения терминологии языков Си / Си++.

Каждая директива препроцессора занимает отдельную строку программы и начинается с символа «#», перед которым может стоять произвольное число пробельных символов. Далее сразу должно находиться имя директивы препроцессора:

- директивы «#define» и «#undef» — определение и отмена определения символов условной трансляции;
- директивы «#if», «#elseif», «#else» и «#endif» — условная трансляция секций текста программы;

- директива «#line» — управление нумерацией строк, использующейся в сообщениях об ошибках и предупреждениях;
- директивы «#error» и «#warning» — генерация пользовательских ошибок и предупреждений;
- директивы «#region» и «#endregion» — дополнительная пометка секций текста программы.

Директивы препроцессора, лежащие в многострочных блочных комментариях, игнорируются. К одной директиве препроцессора, кроме директив «#define», «#undef», «#else» и «#endif», относятся также и строки, идущие следом и начинающиеся символами «возможные пробельные символы # пробел». Содержимое последующих строк после символа «#» присоединяется к первой строке директивы.

Директива «#define имя» определяет имя со значением булевского литерала *true*, а директива «#undef имя» определяет имя со значением булевского литерала *false*. Значение определенного имени доступно только в директивах препроцессора, стоящих ниже по тексту. Значение неопределенного ранее имени равно булевскому литералу *false*. Не накладывается ограничений на любое переопределение указанных имен. Определения действуют до конца текущего файла.

Конструкция условной трансляции задаётся несколькими директивами:

```
#if булевское выражение
  секция текста программы
необязательные ветви #elseif
ветвь #else
#endif
```

Ветвь «#elseif» задаётся следующим образом:

```
#elseif булевское выражение
  секция текста программы
```

Ветвь «#else» задаётся следующим образом:

```
#else
  секция текста программы
```

Булевским выражением является любое булевское выражение языка Sisal, содержащее имена символов условной трансляции и литералы *true* и *false*. Первое истинное булевское выражение директив «#if» и «#elseif» определяет транслируемую обычным образом секцию текста программы, возможно тоже содержащую вложенные директивы условной трансляции. Если значения всех булевских выражений ложны, то транслируется секция текста программы, лежащая после директивы «#else», если она присутству-

ет. Транслируемая секция программы начинается со следующей после директивы строки или после окончания многострочного блочного комментария, начинающегося на одной строке с директивой препроцессора.

Пропускаемые секции текста программы не обязаны содержать лексически правильный текст. Их просмотр осуществляется только для корректного пропуска вложенных директив условной трансляции, не лежащих в многострочных блочных комментариях.

Директива «#line» имеет следующий синтаксис: «#line номер строки , имя файла», где номер строки и имя файла заданы выражениями языка Sisal целого и строкового типа. Директива меняет нумерацию и имя файла в возможных сообщениях об ошибках и предупреждениях текущего файла, начиная со следующей строки текста программы. Номер строки или имя файла (вместе с запятой) можно опустить для сохранения его предыдущего значения. Если опущен номер строки и номер файла, то восстанавливаются значения по умолчанию, как если бы не было ни одной директивы «#line» до этого.

Директива «#error сообщение об ошибке» генерирует ошибку трансляции с указанным сообщением, заданным выражением строкового типа языка Sisal. Сообщение об ошибке необязательно, и при его отсутствии будет сообщаться лишь о ее местоположении. Директива «#warning» аналогична директиве «#error» за исключением того, что вместо ошибки генерируется предупреждение.

Тексту можно прикрепить пользовательскую пометку директивами:

```
#region необязательная пометка
  секция текста программы
#endregion необязательная пометка
```

Необязательная пометка задается выражением строкового типа языка Sisal и не обязана совпадать в директивах «#region» и «#endregion». Секция текста программы транслируется обычным образом и также может содержать вложенные директивы «#region» … «#endregion».

Ниже схематически приведён разбор директив препроцессора на этапе лексического анализа теста:

```
// Preprocessor directives are parsed
// after lexical analysis and before syntax analysis

PreDirDefUndef: PreDirBegin ('define' | 'undef')
                LexLineWhiteSpace+ LexName PreDirEnd;

// Directives accumulate string that will be parsed later separately
PreDirCont:  PreDirBegin
             ('if' | 'line' | 'error' | 'warning' | 'region' | 'endregion')
```

```
            ~'\n'* '\n'? (PreDirBegin LexLineWhiteSpace ~'\n'* '\n'?)*;

PreDirSimple: PreDirBegin  ('else' | 'endif') PreDirEnd;

fragment PreDirBegin: {getColumn()==1}? LexLineWhiteSpace* '#';

fragment PreDirEnd: LexLineWhiteSpace*
  (LexLineCommentOrPragma | LexBlockCommentOrPragma | '\n');
```

8. ФОРМАТ FIBRE ДЛЯ ВНЕШНИХ ЗНАЧЕНИЙ

Формат Fibre описывает синтаксис языка взаимодействия с функцией main на языке Sisal. Формат Fibre описывает текстовое представление типов языка Sisal, которые могут использоваться функцией «main» на языке Sisal. Описание приводится в терминах грамматики ANTLR v3, как и в разделе 7, и лежит в классе LL_1 [14] языков (описание неопределённых здесь лексем находится в разделе 7.3).

```
tokens {
  LexColon = ':';  LexComma = ',';
  LexDots  = '..'; LexGT    = '>';
  LexLCPar = '{';  LexLPar  = '(';
  LexLSPar = '[';  LexLT    = '<';
  LexMinus = '-';  LexNE    = '!=';
  LexPlus  = '+';  LexRCPar = '}';
  LexRPar  = ')';  LexRSPar = ']';

  Lex_error = 'error';  Lex_false = 'false';
  Lex_nil   = 'nil';    Lex_true  = 'true';
}

fibre_unit: (value)*;

value: simple_value | compound_value | 'error';

simple_value:   ('+' | '-') (LexIntConst | LexRealConst) | LexCharConst |
                 'false' | 'true' | 'nil';

compound_value: array | stream | record | union;

array:  '[' ( range+ ':' value+  )? ']' | LexStr;

range:  LexIntConst '..' LexIntConst;

stream: '{' value* '}';

record: '<' value+ '>';

union:  '(' LexIntConst ':' value ')';
```

```
// Whitespace is ignored
LexWhiteSpace:   '\t' | '\n' | '\000B' /*\v*/ | '\f' | '\r' | ' ';

// Comments are ignored.
LexLineComment:  '/' '/' ~'\n'* '\n'?;
LexBlockComment: '/' '*' (options {greedy=false;}: .)* '*' '/';
```

Формат Fibre достаточно очевиден и не требует особых пояснений. Ключевые слова чувствительны к регистру. Ключевое слово «*error*» задаёт ошибочное значение любого типа. Число в представлении объединения обозначает порядковый номер (начинающийся с единицы) задаваемого тега. Значения элементов массива перечисляются в row-major порядке. Не заданные элемента массива и записи предполагаются равными ошибочным значениям (выводится предупреждение). Лишние элементы массива и записи игнорируются и вызывают предупреждение.

9. СТРУКТУРА МОДУЛЯ НА ЯЗЫКЕ СИ++

Конечной целью трансляции модуля на языке Sisal является единица компиляции на языке Си++, удовлетворяющая определённым правилам, описанным в данном разделе.

9.1. Заголовочный файл «sisal.hpp»

В подключаемом заголовочном файле «sisal.hpp» описываются типы языка Sisal на языке Си++, заданные с помощью шаблонов классов. Все объявления заголовочного файла «sisal.hpp» расположены в пространстве имён «Sisal».

Для простых встроенных типов языка Sisal на языке Си++ определяется класс пустого типа Null, булевского типа Boolean, символьного типа Character, целого типа Integer и вещественного типа Real. Данные классы (кроме класса Null) определяют методы error и value. Метод error возвращает значение типа *bool*, равное значению *true*, если значение языка Sisal является ошибочным, и *false* иначе. Метод value возвращает значение *bool* для класса Boolean, значение SisalCharacterType[63] для класса Character, значение SisalIntegerType[64] для класса Integer и значение SisalRealType[65] для класса Real.

[63] Тип SisalCharacterType определяется как *typedef* от некоторого символьного типа языка Си (скорей всего от типа wchar_t).

[64] Тип SisalIntegerType определяется как *typedef* от некоторого целого типа со знаком языка Си (скорей всего от типа *int*).

Встроенные составные типы языка Sisal определяются через шаблоны классов языка Си++. Для классов, соответствующих встроенным типам языка Sisal, в заголовочном файле определены все встроенные операции.

Тип потока задаётся шаблоном класса «Stream<**T**>», где параметр шаблона **T** задаёт тип элементов потока. Тип массива со свободной формой задаётся шаблоном класса «Array**N**<**T**>», где число **N** является частью имени шаблона и задаёт размерность массива, а параметр шаблона **T** задаёт тип элементов массива. Тип массива с фиксированной формой задаётся шаблоном «Array_$\mathbf{D_1}$_$\mathbf{D_2}$_..._$\mathbf{D_N}$<**T**>», где числа $\mathbf{D_1}$, …, $\mathbf{D_N}$ задают количество элементов в соответствующих размерностях массива. Шаблоны класса массива поддерживают операцию получения элемента по его возможно многомерному индексу. Объявить тип массива со свободной формой можно макросом «SISAL_DECLARE_ARRAY(**T**, **N**)». Объявить тип массива с фиксированной формой можно макросом «SISAL_DECLARE_FIXED_ARRAY(**T**, $\mathbf{D_1}$, …, $\mathbf{D_N}$)», где число **N** в текущей реализации ограничено[66] числом 100.

Тип записи задаётся шаблоном класса «Record<$\mathbf{T_1}$, …, $\mathbf{T_n}$[67]>», где типы $\mathbf{T_1}$, …, $\mathbf{T_n}$ задают типы полей записи. Тип объединения задаётся шаблоном класса «Union<$\mathbf{T_1}$, …, $\mathbf{T_n}$>», где типы $\mathbf{T_1}$, …, $\mathbf{T_n}$ задают типы тегов объединения. Доступ к полям записи и тегам союза осуществляется, соответственно, с помощью методов GetField**i** и GetTag**i**, где число **i** лежит в отрезке от **0** до **n-1**.

Тип функции задаётся шаблоном класса «Function<Tuple<$\mathbf{A_1}$, …, $\mathbf{A_n}$>, Tuple<$\mathbf{R_1}$, …, $\mathbf{R_m}$> >», где типы $\mathbf{A_1}$, …, $\mathbf{A_n}$ задают типы формальных параметров функции, а типы $\mathbf{R_1}$, …, $\mathbf{R_m}$ задают типы возвращаемых значений. Шаблон класса «Tuple<$\mathbf{T_1}$, …, $\mathbf{T_n}$>» используется для задания кортежей значений, доступ к элементам которых предоставляет метод GetItem**i**, где число **i** лежит в отрезке от **0** до **n-1**. Шаблон класса функции поддерживает операцию вызова функции.

---

[65] Тип SisalRealType определяется как *typedef* от некоторого вещественного типа языка Си (скорей всего от типа *double*).

[66] В силу невозможности задания макроса с переменным количеством параметров, необходимо было остановиться на каком-то числе.

[67] Число **n**, в силу ограничений реализации на количество параметров шаблона в некоторых компиляторах языка Си++ может не превосходить числа 26. В любом случае, в силу невозможности задания шаблона с переменным количеством параметров, необходимо было остановиться на каком-то числе.

9.2. Определения и объявления типов

Определения переименованных типов языка Sisal соответствуют *typedef* объявлениям типов на языке Си++ со строкой «s_» в начале имени переименованного типа. Определения пользовательских типов соответствуют определениям классов со строкой «s_» в начале имени пользовательского типа, наследуемых от класса базового типа. Объявления пользовательских типов с параметрами соответствуют определениям шаблонов класса со строкой «s_» в начале имени пользовательского типа с параметрами, наследуемыми от класса базового типа.

9.3. Определения и объявления процедур

Объявлениям и определениям функций языка Sisal соответствуют объявления и определения функций языка Си++ со строкой «s_» в начале имени и суффиксом, призванным отличать функции неоднозначные по типам возвращаемых значений. Объявлениям и определениям операций языка Sisal соответствуют объявления и определения операций языка Си++ с теми же знаками, за исключением операций со знаками «**», «. имя» и операций преобразования типов. Операции со знаком «**» соответствует функция с именем «op_pow». Операции со знаком «. имя» соответствует метод с указанным именем в классе, соответствующем пользовательскому типу, являющемуся типом формального аргумента операции. Операции явного преобразования типов соответствует функция с именем «op_cast». Операции неявного преобразования типов, результатом которой является пользовательский тип, соответствует конструктор класса, соответствующего указанному пользовательскому типу. Операции неявного преобразования типов, результатом которой не является пользовательский тип, соответствует операция преобразования в этот тип, расположенная в классе, являющимся типом формального аргумента операции.

Объявлениям и определениям обобщенных процедур соответствуют шаблоны функций (со строкой «s_» в начале имени) и операций (с учетом указанных выше исключений). Функции и операции языка Си++, задающие процедуры языка Sisal, берут на вход и возвращают шаблон класса «Tuple<$\mathbf{T_1}$, …, $\mathbf{T_n}$>».

СПИСОК ЛИТЕРАТУРЫ

1. **McGraw J. R.** Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.2. / McGraw J. R., Skedzielewski S. K., Allan S. J., Oldehoeft R. R., Glauert J., Kirkham C., Noyce B. and Thomas R. — Livermore, CA, 1985. — (Tech. Rep. / Lawrence Livermore National Laboratory; M-146, Rev. 1).
2. **Стасенко А. П.** Транслирующие компоненты системы функционального программирования SFP / Глуханков М. П., Дортман П. А., Павлов А. А. и Стасенко А. П. // Современные проблемы конструирования программ. — Новосибирск, 2002. — С. 69–87.
3. **Cann D. C.** Retire Fortran?: a debate rekindled // Communs. of the ACM. — New York: ACM Press, 1992. — Vol. 35, No. 8. — P. 81–89.
4. **Cann D. C.** Sisal Reference Manual: Language Version 2.0 / Cann D. C., Feo J. T., Böhm A. P. W. and Oldehoeft R. R. — Livermore, CA, 1991. — 128 p. — (Tech. Rep. / Lawrence Livermore National Laboratory; UCRL-MA-109098).
5. **Feo J. T.** Sisal 90 user's guide / Feo J. T., Miller P. J., Skedzielewski S. K. and Denton S. M. — Livermore, CA: Lawrence Livermore National Laboratory, Draft 0.96, 1995. — 80 p.
6. **Бирюкова Ю. В.** Sisal 90: Руководство для пользователя. — Новосибирск, 2000. — 84 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 72).
7. **ISO 7185:1990(E).** Information technology: Programming languages: Pascal. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1990.
8. **Касьянов В. Н.**, **Бирюкова Ю. В.** и **Евстигнеев В. А.** Функциональный язык Sisal // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск, 2001. — С. 54–67.
9. **Стасенко А. П.**, **Синяков А. И.** Базовые средства языка Sisal 3.1. — Новосибирск, 2006. — 60 с. — (Препр. / РАН. Сиб. отд-ние. ИСИ; № 132).
10. **Стасенко А. П.** Внутреннее представление системы функционального программирования Sisal 3.0. — Новосибирск, 2004. — 54 с. — (Препр. / РАН. Сиб. отд-ние. ИСИ; № 110).
11. **ISO/IEC 14882:2003(E).** Programming languages: C++. — Geneva: International Organization for Standardization (ISO), Central Secretariat, 2003.
12. **ANSI X3.4:1986.** Information systems: coded character sets: 7-Bit American national Standard Code for Information Interchange (7-Bit ASCII). — NY: American National Standards Institute (ANSI), 1986.
13. **ANSI/IEEE 754-1985.** IEEE standard for binary floating-point arithmetic. — NY: Institute of Electrical and Electronics Engineers, 1985 (Reprinted in SIGPLAN Notices, 22(2): 9–25, 1987).
14. **Parr T.** The Complete Antlr Reference Guide. — Pragmatic Bookshelf, 2007. — 361 p.
15. **The Unicode Consortium.** The Unicode Standard. — Version 4.0. — Boston, MA: Addison-Wesley, 2003.

16. **ISO/IEC 10646:2003(E).** Information technology: Universal Multiple-Octet Coded Character Set (UCS). — Geneva: International Organization for Standardization (ISO), Central Secretariat, 2003.
17. **Робинсон У.** C# без лишних слов / Пер. с англ. — М.: ДМК Пресс, 2002. — 352 с.: ил. (Серия «Для программистов»).

С.С. Крайниковский

ВЕЙВЛЕТ-ОБРАБОТКА ДАННЫХ В ГЕОФИЗИЧЕСКИХ ИССЛЕДОВАНИЯХ СКВАЖИН

ВВЕДЕНИЕ

В процессе разработки и исследования нефтегазовых месторождений широко применяются методы геофизических исследований скважин (ГИС). Эти методы представляют собой каротажные зондирования, когда в скважину опускается прибор и измеряет характеристики на разных глубинах, и последующую обработку полученных данных. Одними из широко используемых методов остаются метод высокочастотных индукционных каротажных изопараметрических зондирований (ВИКИЗ) и метод бокового каротажного зондирования (БКЗ). В лаборатории электромагнитных полей Института нефтегазовой геологии и геофизики СО РАН разработан прибор ВИКИЗ [1,2], а также многочисленные алгоритмы интерпретации полученных данных. Часть из них реализована в программных системах, таких как МФС ВИКИЗ и разрабатываемая система EMF Pro. Общая схема интерпретации, используемая в этих системах, такова: на первом этапе каротажная кривая анализируется, происходит обработка данных и выделяются пласты — участки с однородными характеристиками сигнала. Затем в полученных пластах строятся осесимметричные n-слойные модели, указывающие на распределение зон проникновения фильтрационного раствора и значение электрического сопротивления. Эти модели затем визуализируются в графическом интерфейсе и служат информативным признаком для специалиста, работающего с данными.

Для обеспечения правильного построения моделей и приемлемой скорости вычислений очень важной является задача первоначальной обработки, так как в результатах измерений часто присутствует аппаратурный и геологический шум, усложняющий анализ данных. Этот шум представляют собой высокочастотные колебания сигнала, которые чаще всего стремятся сгладить фильтрационными преобразованиями кривой. В этот же класс задач входит и корректная расстановка геологических пластов. Корректность определяется геоэлектрической теорией, а также эмпирически накопленным опытом специалистов-геофизиков, которые часто производят расста-

новку границ пластов вручную, исходя из изображения кривых на диаграмме.

1. ЗАДАЧА ВЫДЕЛЕНИЯ ПЛАСТОВ

Рассмотрим процесс построения алгоритма обработки данных. Для того, чтобы понять, какой же результат мы стремимся получить, необходимо построить математическую модель обработки. Так как источником информации, как было сказано выше, являются не только теоретические модели среды, но и эмпирический опыт специалистов, то необходимо разработать ряд критериев, по которым обработка данных считалась бы успешной. В результате совместной работы со специалистами — геофизиками в задаче разбиения на пласты были получены следующие критерии:

Пусть вещественная непрерывная функция $f(z)$ представляет значения сигнала от глубины.

1. Пластом считается участок постоянства сигнала с заданной точностью.
2. Граница пластов пролегает там, где изменение сигнала с глубиной наибольшее, то есть где достигается максимум модуля производной.
3. Так как реальные сигналы не бывают строго постоянными, то постоянство в данном случае принимается с определённой степенью точности δ и оценивается на интервале глубины: если $\forall z \in (a,b) \mid f(z) - c \mid \leq \delta$, где $f(z)$ — анализируемый сигнал, с — среднее значение на интервале, равное $\frac{1}{b-a}\int_a^b f(z)dz$, $a < b$, то $f(z)$ полагается *относительно постоянной* на интервале (a,b). Разрешение δ определяется опытным путём и зависит только от того, насколько мелкие детали геологического разреза необходимо выделить в конкретной задаче. Оно зависит от конкретных свойств разреза и задачи специалиста на данный момент.
4. Используется дополнительный критерий осреднения по глубине, когда высокочастотные детали (шум) не учитываются, а выделяются лишь главные особенности сигнала. При этом также возможно выделить эмпирический порог ε как размер интервала глубины, в пределах которого значения осредняются. ε определяется по тем же принципам, что и δ.

Опишем математическую постановку задачи расстановки границ пластов. Пусть $f:[a,b]\to R$, где $0<a<b$ — непрерывная функция сигнала и $c:[0...N]\subset N\to R$ — её дискретный вариант, который определяется так: $c(0)=a$, $c(i)=f(a+ih)$, где $h\in(0,\infty)$ — шаг дискретизации, $a+Nh<b$, $a+(N+1)h\geq b$. Обозначим $x_i=c(i)$. *Разбиением* интервала глубины назовём множество точек $a=r_0<r_1<...<r_k=b$, которое удовлетворяет некоторым из вышеописанных критериев. Существующие алгоритмы расстановки границ в МФС ВИКИЗ, описанные в [3], основаны на критерии 2) и работают с функцией вертикального разрешения, которая является производной значений зондов, предварительно обработанных с учётом специфики геофизического прибора. Границам r_i соответствуют локальные максимумы этой функции. В практической реализации алгоритм работает с дискретными значениями сигнала; вычисляются разностные производные.

Использование данного подхода сопряжено с трудностями и ограничениями, основными из которых является большая зашумлённость данных, характеризующаяся высокочастотными колебаниями сигнала. Кроме того, не учитывается критерии 3) и 4), когда алгоритм не может отличать низкоамплитудные колебания от больших скачков, а также отсеивать мелкие частоты.

В результате работы алгоритма на зашумлённых данных мощность пласта близка к шагу дискретизации кривой, пласт содержит порядка 1–2 точек, что неприемлемо с точки зрения интерпретационных алгоритмов и не отражает свойств геологического разреза.

На данный момент используется фильтрация данных, которая перед применением алгоритма позволяет сглаживать значения и избавляться от мелких несущественных деталей.

Фильтрация представляет собой функцию f: $[x_1...x_n]\to[y_1...y_n]$, которая определяется так: $y_i=\mathrm{f}(\mathrm{x}_i)=\sum_{j=-k}^{k}a_jx_{i+j}$, где $a_j\in\mathrm{R}$, $[x_1...x_n]$ — вектор значений каротажной кривой. Обычно вектор коэффициентов $[a_{-k}\ldots a_k]$ симметричен относительно a_0, и a_0 больше по модулю остальных значений, $a_i>0$, $\sum_{i=-k}^{k}a_i=1$. Данное преобразование представляет собой осреднение с весовыми коэффициентами и в какой-то мере решает задачу фильт-

рации по глубине и использования критерия 4), так как сглаживает детали сигнала с характерным размером по глубине менее 2k. Однако сравнение работы алгоритмов расстановки границ экспертами-геофизиками при использовании фильтрации и без неё показывает, что существенного улучшения качества разбиения геологического разреза не происходит.

В постановке задачи разбиения разреза на пласты, основанной на критериях 1) и 3) для любого интервала (r_i, r_{i+1}) из разбиения выполнены условия относительного постоянства f с погрешностью ε. Для того, чтобы разработать алгоритм, наиболее полно учитывающий все критерии, было решено исследовать сигналы с помощью вейвлет-преобразований, которые могут дать информацию как об амплитудной, так и о частотной составляющей геофизического сигнала, а также позволяют гибко решать задачи осреднения данных.

2. ВЕЙВЛЕТ-ПРЕОБРАЗОВАНИЯ И ИХ ПРИМЕНЕНИЕ К ЗАДАЧЕ ВЫДЕЛЕНИЯ ПЛАСТОВ

Идея вейвлет-преобразования сигнала в общем случае заключается в том, чтобы представить функцию из некоторого класса в виде разложения её по базисным функциям, подробнее об этих преобразованиях и их приложениях можно узнать из [4].

В технологических задачах обработки сигналов часто используется кратномасштабный вейвлет-анализ, описанный ниже. Исследуемая функция представляет собой конечный набор вещественных значений, что позволяет представить её как ступенчатую функцию с одинаковой длиной ступенек: $f_0(t) = c_i$ на интервале $[ih, (i+1)h[, i = 0...2^N - 1, h \in]0, \infty[$. Все функции такого вида, определённые на конечном полуинтервале, обозначим как пространство V_0. Аналогично можно рассмотреть пространство V_1 функций, определённых на том же полуинтервале, но ступеньки будут в 2 раза шире: $d_{k,i}$' Таким образом, можно получить последовательность вложенных пространств $V_m \subset V_{m-1} \subset ... \subset V_0$, где $m \le n$. Легко установить, что функции вида $\phi_{m,i} = \chi_{[i*2^m,(i+1)*2^m[}$, где $i = 0...2^{N-m}$, $\chi_{[a,b[}$ — характеристическая функция полуинтервала $[a,b[$, составляют базис соответствующего пространства V_m. Вводятся также дополнительные функции $\psi_{m,i}(x) = 1$, если

$x \in \left[i*2^m, (i+1)*2^m / 2 \right[$ и $\psi_{m,i}(x) = -1$, если $x \in \left[(i+1)*2^m / 2, (i+1)*2^m \right[$, и равные нулю в остальных случаях. Основным утверждением, используемым в обработке дискретных сигналов, является существование разложения сигнала по базисным функциям разных уровней:

$$f_0(x) = \sum_{i=0}^{N/2^m} c_{m,i}\phi_i(x) + \sum_{i=0}^{N/2^m} d_{m,i}\psi_{mi} + \sum_{i=0}^{N/2^{m-1}} d_{m-1,i}\psi_{m-1,i} + \ldots + \sum_{i=0}^{N/2} d_{1,i}\psi_{1,i}$$

С учётом этого сигнал можно расценивать как сумму функций, каждая из которых отражает вклад различных частотных составляющих. Суммы, соответствующие функциям $\psi_{k,i}$ отражают «вклад» частот характерным размером длины 2^k.

Рассмотрим процедуру преобразования сигнала с помощью методов трешолдинга коэффициентов, подробное описание которого можно найти в [5]. Преобразование представляет собой функцию $tr_c : R^+ \to R^+$, определяемую так: $tr_c(x) = x$, если $x \geq c$ и $tr_c(x) = 0$ иначе; $c \geq 0$. Функция tr_c применяется к коэффициентам $d_{k,i}$, участвующим в разложении сигнала на подпространства, в результате получаются новые коэффициенты, которые будем обозначать как $d_{k,i}'$.

Рассмотрим функцию $f_0^{'}$, которая получается из аналогичной формулы разложения по подпространствам, но с помощью изменённых коэффициентов:

$$f_0^{'}(x) = \sum_{i=0}^{N/2^m} c_{m,i}\phi_i(x) + \sum_{i=0}^{N/2^m} d_{m,i}'\psi_{m,i} + \sum_{i=0}^{N/2^{m-1}} d_{m-1,i}'\psi_{m-1,i} + \ldots + \sum_{i=0}^{N/2} d_{1,i}'\psi_{1,i} .$$

Отметим некоторые свойства $f_0^{'}$:

1) $f_0^{'}$ является ступенчатой функцией как линейная комбинация ступенчатых базисных функций.

2) Рассмотрим норму $\|f\|_\infty = \max_{x \in dom(f)} |f(x)|$. Тогда $\|f_0 - f_0'\| \leq cm$, где m — количество уровней разложения.

3) На любом постоянном полуинтервале значение с функции f_0' является средним значением f_0 на этом же полуинтервале.

Полученные свойства дают основания для использования данных видов преобразований в задаче разбиения геологического разреза.

– Решение 1: Пусть f_0 — функция геофизического сигнала, f_0' — преобразование, полученное с помощью трешолдинга коэффициентов. Значения границ пластов r_i положим равными границам «ступенек», то есть точкам f_0', соседние значения которых различны. Это решение обеспечивает выполнение критериев 1) и 3). Для заданной точности ε $c \le \frac{\varepsilon}{m}$ согласно свойству 2).
– Решение 2: Подвергнем коэффициенты $d_{k,i}'$ дополнительному преобразованию, которое зануляет все коэффициенты с индексами k меньше фиксированного значения l. В таком случае происходит удаление всех деталей, характерный размер которых менее $\delta = 2^{l-1}$, что позволяет утверждать о выполнении критерия 4).

На основе этих решений были проведены эксперименты с различными параметрами c и уровнями l, реализованные с помощью пакета MathLab, часть из которых была оценена экспертами — геофизиками.

На рисунках, приведённых ниже, изображены результаты работы алгоритма со следующими параметрами: зануление коэффициентов уровня 1, трешолдинг коэффициентов со значениями c =5, 15; m = 7. На горизонтальной оси отображается глубина в метрах, на вертикальной — показания приборов в виде разности фаз, в градусах. Анализ графиков показывает, что при меньших значениях коэффициента c пласты разбиваются более мелко, но приближают с хорошей точностью, в то время как при больших значениях коэффициента c выделяются всё более крупные пласты, но теряются мелкие особенности сигнала.

Другой эксперимент заключался в вариации уровня зануления l = 1 и l = 4 (что соответствует осреднению деталей характерной длиной 2 и 16 точек соответственно). На диаграммах, приведённых ниже, это соответствует 0.2 и 1.6 м.

Очевидно, что во втором случае теряются высокочастотные детали, представляющие единичные всплески и выбросы. При этом c = 5, как и прежде.

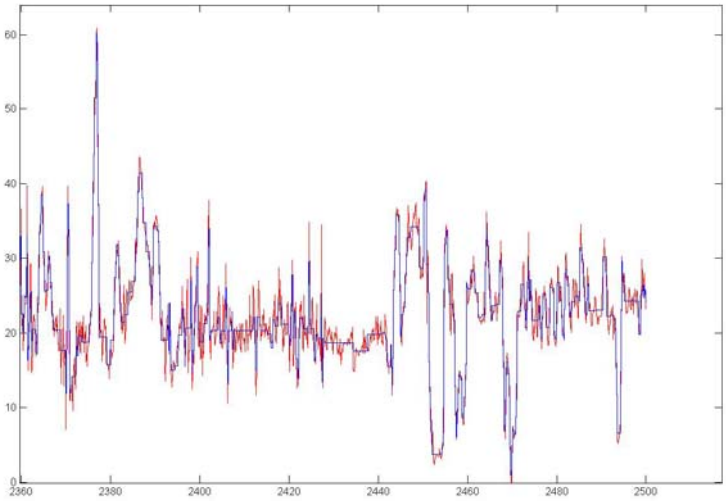


Рис. 1. Применение вейвлет-преобразования Хаара к каротажным данным. Коэффициент c = 5

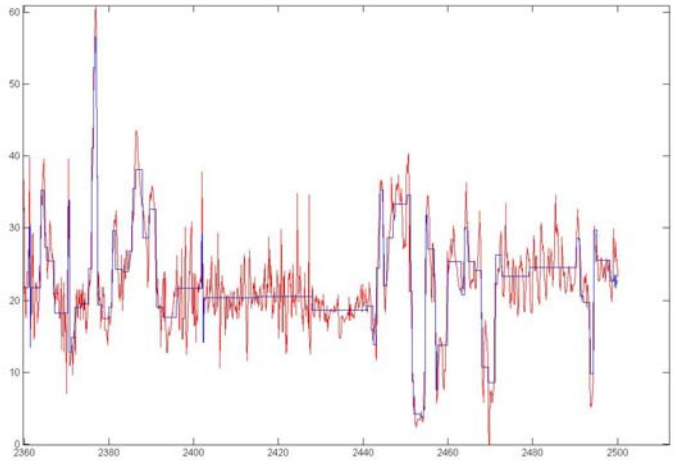


Рис. 2. Применение вейвлет-преобразования Хаара к каротажным данным. Коэффициент c = 15.

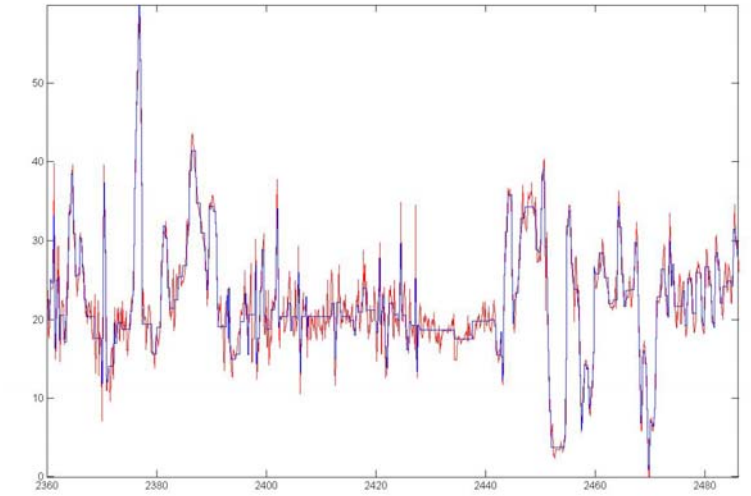


Рис. 3. Применение преобразования Хаара с удалением деталей уровня 1

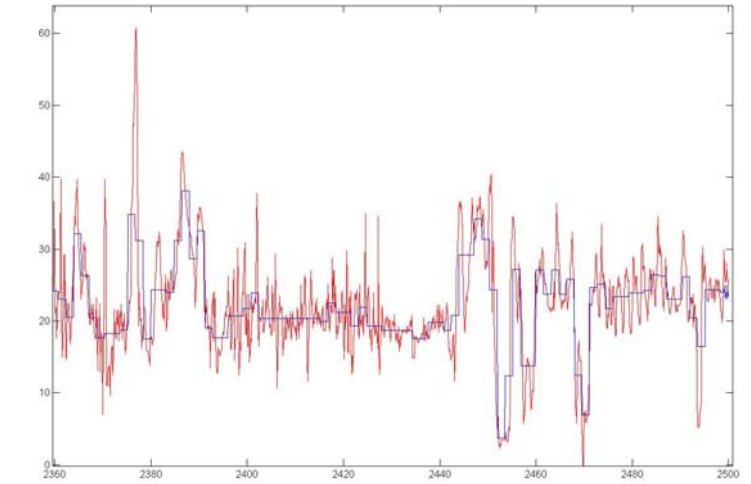


Рис. 4. Применение преобразования Хаара с удалением деталей уровня 4

3.ОСНОВНЫЕ РЕЗУЛЬТАТЫ И ВЫВОДЫ

Надо заметить, что оценка реальной эффективности работы алгоритмов в наибольшей степени формируется его использованием специалистами в геофизической практике, поэтому полученные результаты показывались специалистам-геофизикам, в результате чего были сделаны следующие замечания:

1) использование вейвлет-преобразований Хаара в большинстве случаев даёт более достоверную картину, чем прежний алгоритм, описанный в [3];
2) алгоритм, основанный на вейвлет-преобразованиях позволяет более гибко настраивать параметры обработки кривых;
3) недостатком может являться то, что длина пласта всегда есть число, кратное $2^k h$, что вносит некоторую «машинную составляющую» в картину разреза. Однако для работы алгоритмов интерпретации этот фактор не является существенным, либо может быть устранён при доработке алгоритма.

Перспективой развития данного подхода является разработка программных модулей, позволяющих настраивать параметры преобразований в пользовательском интерфейсе, а также усовершенствование самого алгоритма. Также важно установить, как будет формироваться исследуемая функция из показаний различных зондов прибора ВИКИЗ и провести более детальные оценки эффективности.

СПИСОК ЛИТЕРАТУРЫ

1. Антонов Ю.Н., Жмаев С.С., Большаков В.И., Кисилев В.В., Мышлявцев А.В. Устройство для каротажного электромагнитного зондирования. Авторское свидетельство №1004940 (СССР). Бюл. изобр. № 10, 1983 г.
2. Снопков В.П., Антонов Ю.Н., Жмаев С.С., Киселев В.В. Устройство для электромагнитного каротажного зондирования. Патент РФ № 2092875. Бюл. изобр. № 28, 1997 г., 6 G 01 V 3/28, 3/18, 3/30.
3. Технология исследований нефтегазовых скважин на основе ВИКИЗ. Методическое руководство / Под ред. Эпов М. И, Антонов Ю. Н. — Новосибирск: Изд-во СО РАН, НИЦ ОИГГМ, 2000. — 121 с.
4. Воробьев В.И., Грибунин В.Г. Теория и практика вейвлет-преобразования. — С.-Пб.: Изд-во Военного университета связи, 1999.
5. Алексеев К.А. Теория и практика шумоподавления в задаче обработки сейсмоакустических сигналов, материалы сайта http://matlab.exponenta.ru/

С. С. Крайниковский

РАЗРАБОТКА ГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ И ВИЗУАЛИЗАЦИЯ ДАННЫХ В ГЕОФИЗИЧЕСКИХ ПРОГРАММНЫХ СИСТЕМАХ

В процессе разработки нефтегазовых месторождений широко применяются методы геофизических исследований скважин (ГИС). Процесс исследования включает в себя каротаж — измерение электрических и электромагнитных характеристик околоскважинной среды на различных глубинах и интерпретацию полученных данных. Для проведения каротажа используются различные приборы и методы, такие, как, например, ВИКИЗ (высокочастотное индукционное каротажное изопараметрическое зондирование) и БКЗ (боковое каротажное зондирование). Данные методы широко используются на территории России, Китая и некоторых стран СНГ. Процесс интерпретации полученных данных заключается в том, чтобы по измеренным каротажным кривым определить характер залегания пород, и в конечном итоге, получить информацию о распределении углеводородного сырья. Для проведения интерпретации применяются программные системы, такие, как, например, МФС ВИКИЗ — система интерпретации метода ВИКИЗ, разработанная в Институте нефтегазовой геологии и геофизики СО РАН.

Деятельность по интерпретации можно разделить на две важные составляющие. Первая из них — визуальная оценка специалистом графиков и диаграмм с целью определения особенностей кривых и выделения геологических пластов. Вторая — построение различных моделей среды и решение этих задач методами вычислительной математики. Несмотря на развитие теории геофизического моделирования и совершенствование алгоритмов, первая составляющая играет очень важную роль. Поэтому одним из основных требований к программным системам является удобный и многофункциональный пользовательский интерфейс с широкими возможностями визуализации данных.

Интерфейс программной геофизической системы представляет собой совокупность графических управляющих элементов (кнопок, меню, окон и т.д.), а также различные графики, диаграммы, таблицы, трёхмерные сцены — то, каким образом представляются данные.

Процесс разработки графических интерфейсов заключается в систематизации требований, созданию проекта интерфейса, примеров работы (демо-версий, слайдов и т.д.), а также обратной связи, когда специалисты — потенциальные пользователи оценивают работу. На этом этапе идёт оценка удобства, информативности, могут добавляться новые требования и изменяться старые.

Рассмотрим процесс создания графического интерфейса программной системы EMF Pro, разработанной в Институте нефтегазовой геологии и геофизики СО РАН. Основной функциональностью системы является автоматическая интерпретация каротажных данных методов ВИКИЗ и БКЗ с помощью пластовых осесимметричных n-слойных моделей. В системе собраны алгоритмы построения стартовых моделей, решения прямой и обратных задач [1], совместной интерпретации, двумерной задачи и т.д. Имеется набор фильтраций, позволяющий обрабатывать данные до начала работы основных интерпретационных алгоритмов. Эти особенности отражаются в таком требовании к интерфейсу, как логичная и удобная схема работы с множеством методов и алгоритмов. В проекте реализована концепция «одни данные — много представлений», когда, например, одни и те же каротажные кривые можно посмотреть в разных диаграммах, с разными настройками. Она позволяет осуществлять прозрачное и удобное управление информацией.

Другим важным требованием является соблюдение традиционных для геофизика типов представления данных. Каротажные кривые и модели решено было отображать с помощью макетов (Templates). Макет представляет собой окно с размещёнными на нём диаграммами, по вертикальной оси которых обычно отображается глубина, а по горизонтальной — шкала отображаемой величины. Макет также включает в себя заголовки и подписи. Пример разработанного макета для EMF Pro изображён на рис. 1.

Следующая важная задача, встающая при разработке интерфейсов, заключается в выборе типов визуализации данных. Цель — определить, какие из примеров наиболее информативны для специалиста-геофизика. Традиционным представлением считаются каротажные кривые, изображённые на графике (рис. 2).

Были опробованы примеры альтернативного представления каротажных данных ВИКИЗ: на рис. 3 изображена карта изолиний, на горизонтальной оси которой отображается глубина, на вертикальной — номер зонда ВИКИЗ.

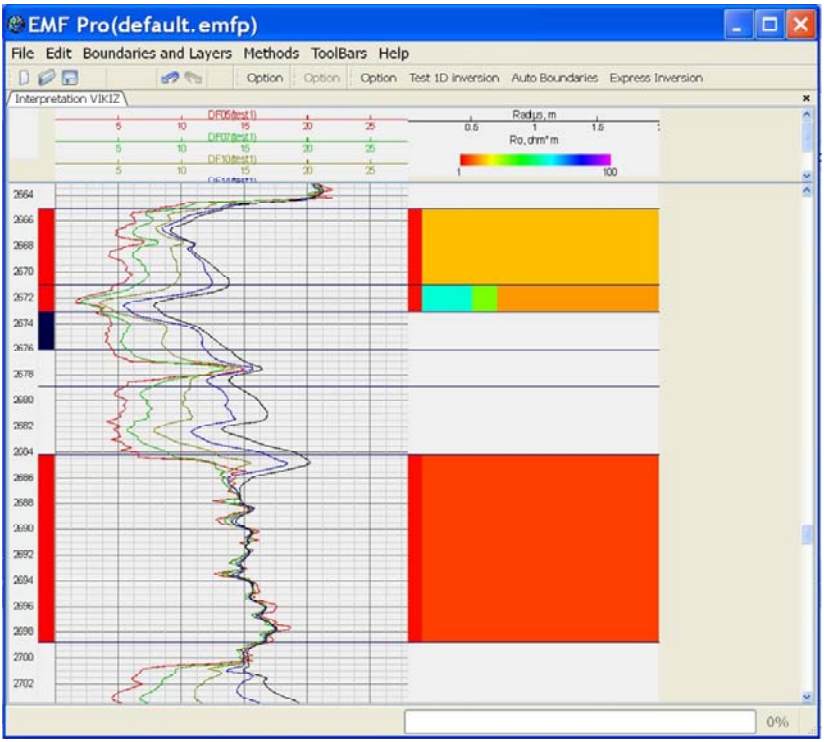


Рис. 1. Пример макета для графического интерфейса системы EMF Pro

Диаграмма позволяет выявить относительно однородные участки геологического разреза и выделить границы пластов.

Следующий пример тех же самых данных изображён на рис. 4 и представляет собой поверхность с источником света. Это изображение позволяет выявлять нарушения монотонности между различными показаниями зондов; здесь они выделяются тенями.

Пример, изображённый на рис. 5, представляет некоторую вариацию предыдущей поверхности, но выполненную в параллельной, а не перспективной проекции. Она также хорошо позволяет выделять нарушения монотонности между различными зондами на одинаковой глубине. Построение проводилось при помощи программы Surfer.

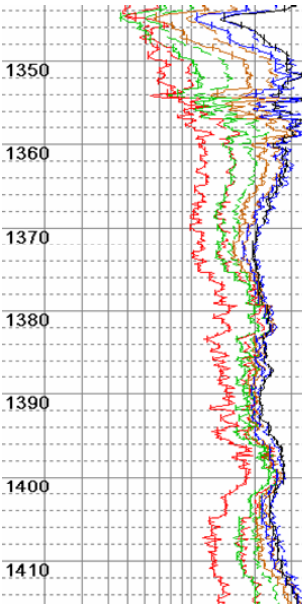


Рис. 2. Традиционное представление каротажных кривых

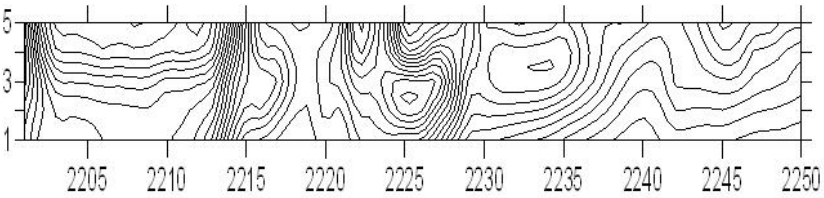


Рис. 3. Карта изолиний диаграммы ВИКИЗ

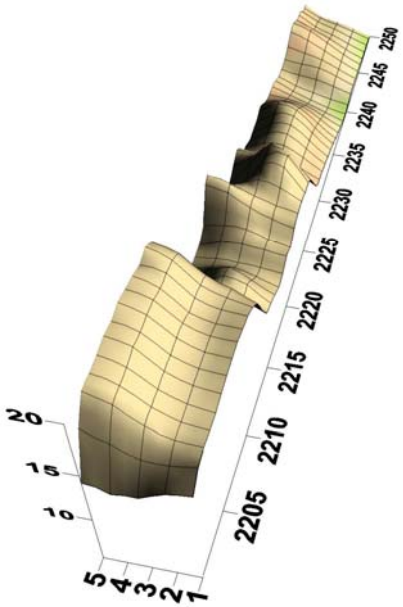


Рис. 4. Поверхность с источником света, визуализирующая данные зондов ВИКИЗ

Таким образом, был построен ряд примеров, и получены оценки от геофизиков. Наилучшими были признаны представления с источником света. Это позволяет включить соответствующие представления (при определённой доработке и модификации) в программные геофизические системы.

В заключение необходимо отметить, что данные результаты исследований по визуализации, проводимых в рамках проекта EMF Pro, предполагается сделать частью функциональности программной системы и внедрить в промышленное использование.

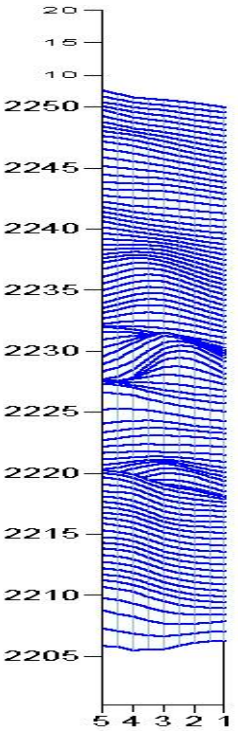


Рис. 5. Параллельная проекция поверхности

СПИСОК ЛИТЕРАТУРЫ

1. Эпов М.И., Авдеев А.В, Горбенко Н.И., Ельцов И.Н., Лаврентьев М.М. Быстродействующие алгоритмы обработки данных электромагнитного каротажа нефтяных скважин // Технологии ТЭК, апрель 2005. — С. 99–105.

П. А. Марчук

ИСПОЛЬЗОВАНИЕ НЕСПЕЦИФИЧЕСКИХ ОНТОЛОГИЙ ДЛЯ ХРАНЕНИЯ ФАКТОГРАФИЧЕСКИХ ДАННЫХ

ВВЕДЕНИЕ

В настоящее время существует множество подходов к созданию информационных ресурсов. Наиболее распространённым подходом является создание ресурса на базе единого хранилища с общим администрированием и присоединение к нему разделённых данных. Однако для многих целей такой подход неприменим, поскольку требует согласия всех участников быть зависимыми от этого ресурса. При этом данные участников не просто становятся формально доступными для пользования в едином хранилище, но и фактически передаются. В процессе работы новая версия данных может находиться либо в главном хранилище, и тогда за новейшей информацией приходится обращаться к нему, либо, как и в начале, у участника, и тогда в главном хранилище оказываются устаревшие данные. Периодическая синхронизация данных решает проблему их устаревания, однако сама по себе является нетривиальной задачей и не снимает проблему собственности.

При переходе от единого хранилища к распределённой модели хранения данных [1] создаётся единое информационное поле, в котором участники контролируют свои данные, предоставляют к ним доступ и при этом получают возможность расширить свои данные за счёт привлечения данных из смежных областей. Элементы перехода к распределённому хранению данных и его практическое применение для электронных архивов является целью данного исследования.

Уже вошедшая в обиход концепция Resource Description Framework (RDF) [2] предлагает несколько вариантов решения проблемы распределённого хранения данных и возможности их связывания в единую информационную систему (или системы) нового поколения. Однако для учёта многих специфических деталей работы с электронными архивами необходимо дополнить эту концепцию собственной методологией.

С точки зрения программного обеспечения существует несколько решений на базе RDF для доступа к данным, например, Sesame [3] или Jena [4].

В нашей работе эти программные продукты не обеспечивают полноту решения и могут применяться лишь для части подзадач.

1. ПОДДЕРЖИВАЕМЫЕ ДАННЫЕ И МЕСТА ХРАНЕНИЯ

Определим исходные данные, которые должны содержаться в информационной системе.

Первый вид данных — это собственно объекты хранения, т.е. предметы, которые представлены в архивах, библиотеках, музеях и т.д.; например, фото-, видео- и аудиодокументы или ссылки на труды и артефакты. В случае со ссылками первичные объекты — это то, на что они ссылаются.

Второй вид данных — это информация о первичных объектах, метаданные. Метаданные включают названия, описания, идентификаторы, а также связи с другими объектами второго вида. Для каждого первичного объекта создаётся метаобъект, и уже метаобъекты связываются в семантическую сеть [5]. Помимо этого, метаобъекты создаются для объектов реального мира, которые должны быть представлены в информационной системе. Объекты реального мира — это персоны, организации, мероприятия и события, географические объекты и т.д.

Программные продукты для операций с объектами могут различаться. Они могут иметь различные версии, располагаться на разных серверах или виртуальных машинах и нести разную операционную нагрузку. Это могут быть серверы для публичного представления, внутреннего редактирования, резервного хранения, тестирования новых версий онтологий и программного обеспечения и т.п. На каждом из этих серверов или виртуальных машин могут содержаться разные виды данных, конфигурационные файлы и версии программного обеспечения. Однако все они осуществляют поддержку фактографической информационной системы.

2. ИНТЕГРАЦИЯ ДАННЫХ ИЗ РАЗНЫХ ИСТОЧНИКОВ

При формировании информационной системы осуществляется введение имеющейся информации. При этом порой необходимо интегрировать информацию из разных источников. Интеграция информации представляет определённые проблемы, например, могут различаться форматы данных, идентификация объектов, а информация может дублироваться или быть противоречивой.

Первостепенной является задача унификации формата данных. Для этой цели в данной работе применяется модификация интероперабельного стандарта метаданных Dublin Core [7].

Поскольку наиболее важными в контексте нашей системы являются фактографические данные, онтология системы основывается именно на этих параметрах. В качестве базовых понятий взяты следующие типы метаобъектов:

1) персона — это, как правило, человек, хотя может быть и другое живое существо;
2) организационная система — организация, мероприятие, отдел, ассоциация, клуб и т.п.;
3) географическая система —страна, регион, город (с точки зрения места), океан, море, река, локальное природное место и т.п.;
4) документ —текст, фото, видео, аудио и т.п.

Множества, обозначаемые этими понятиями, в принципе могут пересекаться и не позволяют описать весь мир, однако достаточны в качестве фактографической базы для нашей системы.

Онтология описана на языке OWL (Web Ontology Language) [8]. На её основе программные системы могут генерировать интерфейсы для работы с данными. Таким образом возможно получение редактора данных, достаточно универсального, пока онтология описана соответствующим образом, а данные соответствуют онтологии.

Данные в системе хранятся в виде XML-файлов формата RDF. При добавлении новых данных в систему из какого-либо источника возможно создание нового файла для этих данных, что даёт возможность повторной интеграции путём замены непосредственно этого файла и учёта этих данных как данных из конкретного источника.

3. ЭКСПРЕСС ВНЕСЕНИЕ ДАННЫХ В БАЗУ ПО ИМЕЮЩЕЙСЯ ОНТОЛОГИИ

При интегрировании данных в единую информационную систему, помимо источников с той или иной формальной структурой данных возможно столкновение с источником без какой-либо формальной структуры. Например, это может быть текст, из которого можно почерпнуть нужные факты, или знания конкретного человека, которые можно внести в информационную систему в виде данных, полученных из рассказа. Возникает проблема экспресс-ввода метаобъектов.

Ввод информации должен быть оперативным, поскольку данных может поступать много в ограниченный промежуток времени. Например, при внесении данных по рассказу в реальном времени (интервью) задержки при вводе информации могут привести к тому, что человек потеряет мысль или же не успеет рассказать всю интересующую информацию, и она останется незафиксированной. Кроме того, поступление данных без формальной структуры либо данных, не укладывающихся в существующие категории, может существенно задержать и затруднить работу операторов системы.

В связи с этим необходимо в кратчайшие сроки создавать новые метаобъекты. Определим, что представляет собой внесение нового метаобъекта в систему. Как правило, в первую очередь вносится имя этого объекта. Отметим, что имя несущественно для некоторых объектов, таких как, например, фотографии или аудиозаписи, поскольку не представляет фактографического интереса и может генерироваться автоматически при внесении в систему. Наиболее важным является установление для нового объекта как можно большего количества ассоциативных связей при внесении в семантическую сеть. Интерес представляют связи, в которых объект состоит с другими объектами, как ранее внесёнными, так и отсутствующими в системе, но фактографически релевантными с учётом позиционирования конкретного архива, музея или библиотеки.

Рассмотрим ввод новых метаобъектов на примере фотодокументов.

В первую очередь необходимо установить, что отражено на фотографии. Отражение — отношении в онтологии, означающее, что некоторый объект фигурирует в некотором документе. Используя стандартный вид архивной карточки фотографии с учётом имеющейся онтологии, необходимо установить следующие факты:

1) когда это происходит (дата);
2) где это происходит: как правило, город (геосистема), где сделана фотография;
3) что здесь (на фотографии) происходит: например мероприятие (оргсистема);
4) кто здесь отражен: персоны, возможно здание организации (оргсистема), возможно какая-нибудь река на заднем плане (геосистема);
5) кто автор фотографии (персона).

Если при этом тот или иной объект не описан в системе, необходимо сразу же внести его в систему и по возможности описать до перехода к следующей фотографии. Если объект уже присутствует, то для фотографии

необходимо установить с ним ассоциативную связь, а не добавлять его дубликат; соответственно, нужен механизм оперативного поиска объектов.

Проблематика быстрого ассоциирования нового объекта с существующими достаточно интересна. Ниже приведены некоторые из проблем и решения, предложенные в разработанной системе.

Отсутствие четкого формального принципа создания названий. Проблема типична для таких объектов, как мероприятия.

1. Давно прошедшие официальные мероприятия, статус которых оператору трудно установить; например, конференция может называться слётом, симпозиумом или конгрессом.

2. Неофициальные мероприятия, названия у которых отсутствует вообще; например, выезд на природу можно назвать пикником или отдыхом на природе. Фактографически такие мероприятия могут быть весьма значимыми, например, если на них присутствовали руководители стран, крупных организаций и т.п. Спецификацию наименований подобных мероприятий должны разрабатывать специалисты, отвечающие за конкретный информационный ресурс.

Со своей стороны, мы можем помочь этому, во-первых, давая возможность специалистам ввести несколько названий этому мероприятию. В нашей онтологии эта возможность предоставляется при помощи конструкции «именование», позволяющей к существующему метаобъекту добавить псевдонимы. При этом при поиске нужного нам мероприятия, поиск будет осуществляться не только по имени, но и по псевдонимам. Во-вторых, конечно, мы можем использовать и стандартный поиск, позволяющий искать по ключевым словам, по дате и другим свойствам объекта. Используя преимущества нашей онтологии, мы:

1) ищем не абстрактный текстовый документ с такими словами, как это делают стандартные поисковые системы, а объект с соответствующими ключевыми словами в определении или соответствующими свойствами;

2) можем осуществлять более гибкий поиск с использованием связей искомого объекта с другими объектами базы данных (например, «человек, работающий в определенной организации»).

Проблема разных названий может встречаться не только у мероприятий, но и у людей, и у организаций. У людей это, например, смена фамилий, либо псевдонимы. У организаций — сокращенные названия, названия на других языках. При помощи конструкции «именование» и соответствующей организации поиска, мы обнаруживаем подход к решению этой проблемы.

Проблема просмотра всей базы для поиска ассоциируемых объектов. Поиск по все базе данных занимает время, особенно, если запрос для этого поиска вовлекает в себя не только ключевые слова и непосредственные свойства, но и связи с другими объектами. Кроме того, часто документы вводятся из одной серии либо одного года, поэтому для упрощения ввода новых документов программа может помогать пользователю, контролируя контекст новых вводимых документов.

Рассмотрим несколько возможных вариантов контекстного контроля. Самое простое — это статические параметры, которые в данный момент присутствуют у всей вводимой серии. Например, это автор документа либо источник поступления. Реже — дата документа, конкретное мероприятие, которое отражено в этой серии документов.

Контекст не является физической папкой для хранения документов. Контекстные «навески» не обязательно соответствуют виртуальным папкам, которые «навешиваются» на серию объектов. Термин «виртуальная папка» часто используется в нынешних структурах хранения данных, однако это только одно направление структуризации, один вид грани на ассоциативном графе. В нашей онтологии для каждого объекта устанавливается соответствующая ассоциативная связь (например, с автором этого документа), либо указывается конкретное свойство (например, дата). Хотя по каким-то темам эти документы все равно могут быть сгруппированы, исходя из предпочтений информационного специалиста, и в этом случае это будут виртуальные папки («коллекции» в онтологии).

Однако несмотря на то, что хранение контекста происходит при помощи модификации свойств или установления связей для каждого объекта, для ускорения процесса ввода мы должны определить контекст сразу для всей вводимой серии документов.

Помимо названных статических параметров, существуют аналогичные динамические параметры, т.е. те, которые очень похожи в этой серии документов, однако полного совпадения между ними нет. Например, это может быть список отраженных персонажей в документе. Если у нас идет серия документов, то персонажи при этом часто повторяются, и мы снова сталкиваемся с понятием контекста. В качестве простейшего варианта решения мы можем сразу вывести список недавно использованных объектов заданного типа, например, список персон, с которыми недавно была установлена ассоциативная связь. При этом, если мы будем выводить их в порядке убывания времени их последнего «использования», то при минимальных затратах это будет существенно увеличивать скорость ассоциирования документов и персон.

Вкратце обозначим более сложные возможности подбора контекстных динамических параметров. Помимо их недавнего использования в процессе ввода, у нас уже имеется семантическая сеть между другими документами и персонами, между организациями и персонами. К тому же у нас есть хронологическое распределение объектов, географическое распределение объектов и т.д. Исходя из этого при помощи более сложных алгоритмов мы можем осуществлять выборку нужного контекста и ощутимо ускорять ввод новых документов, а также подсказывать пользователю возможные варианты.

4. РАСПРЕДЕЛЕННОЕ РЕДАКТИРОВАНИЕ ДАННЫХ

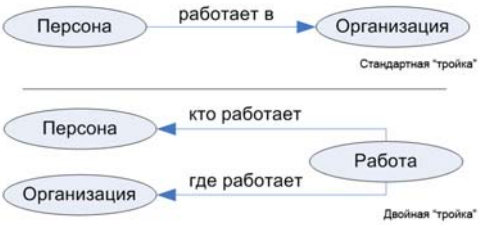
В этом разделе речь идет прежде всего о метаобъектах, которые содержатся в нашей информационной системе. Некоторые из них были внесены в нашу конкретную систему, некоторые могут храниться удаленно (при этом в нужном нам формате). Некоторые могут быть также на другом удаленном сервере и, к тому же, в другом формате, но при этом у нас есть импорт-фильтр для их использования. В общем, исходя из первоначальных источников мы получаем RDF-документы, в которых хранятся метаобъекты. В совокупности эти RDF-документы дают нам единое информационное поле, с которым мы должны работать.

При работе с этими метаобъектами в нашем информационном поле в режиме чтения, если у нас нет проблем с дублированием идентификаторов, вся работа выстраивается достаточно простым способом. Например, мы должны найти объект, чтобы посмотреть его свойства, либо мы должны найти объекты, которые ссылаются на этот объект, чтобы отобразить их свойства. Методология такого поиска объектов, части нашего ассоциативного графа описана, например, в языке поиска SPARQL Query Language for RDF [9].

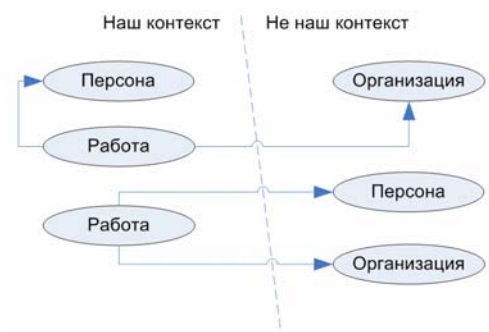
В этом разделе сформулирована проблема, а так же предложен вариант решения этой проблемы в том случае, если нам требуется редактировать информацию в нашем информационном поле, при том, что у нас есть источники, которые находятся не в нашей собственности.

Сначала обозначим некоторую принципиальную для данного раздела особенность нашей онтологии. В стандартной концепции графов RDF ассоциативные связи между объектами обозначаются при помощи «троек» (Triples) RDF. В тройку входит субъект, предикат и объект (см. рисунок, первая часть). В нашей же онтологии базовая схема установления ассоциативной

связи между объектами выглядит следующим образом: ассоциативную связь, которую мы устанавливаем, мы также заводим в качестве метаобъекта в базу данных, и эта связь ссылается на объекты, между которыми мы эту связь устанавливаем (см. рисунок, вторая часть). Фактически одна связь представлена в виде двух «троек».



Теперь определим контекст работы. Зададим простейшую модель, при которой в нашей единой распределенной информационной системе существуют только 2 источника — наш источник, где мы можем редактировать данные, и чужой источник, данные которого мы редактировать не можем, но можем просматривать, а он наши данные ни просматривать, ни редактировать не может. При этом нашей областью просмотра является вся распределенная система, а контекстом редактирования — только наш источник.



Исходя из этой модели и нашей онтологии, мы можем добавлять в наш контекст новые метаобъекты, редактировать их свойства и устанавливать связи между нашими метаобъектами. Кроме того, мы можем устанавливать связи и между объектами разных источников (см. рисунок). При этом, имея областью просмотра все источники, мы получим полную информационную картину. Второй источник полной картины не получит, так как его область просмотра по нашему определению не включает нашего источника, но и противоречий наши добавленные объекты не вызовут.

Так как основу семантической сети составляют скорее связи между объектами, нежели конкретные свойства каждого объекта, то, используя приведенную выше модель, мы достигаем заметного результата, однако полной универсальности данному решению пока недостает.

Например, поскольку мы создаем много новых метаобъектов, а у каждого метаобъекта могут быть несколько свойств, то свойства также начи-

нают играть ощутимую роль. Мы можем захотеть изменить уже имеющееся свойство, например, неправильное название организации, либо добавить новое свойство, например, дату основания.

Кроме того, мы можем посчитать, что какой-то метаобъект неверен, например, человек никогда не работал в такой-то организации, а в базе данных есть соответствующее упоминание. Таким образом, мы должны удалить какой-то метаобъект. Понятно, удалить его из своего контекста не составляет труда, однако это становится проблемой в случае с чужим контекстом.

В случае с главной централизованной базой данных данная проблема решается на уровне полномочий. Пользователю дается право на редактирование определенной области данных; возможно, после редактирования это действие должно быть подтверждено модератором соответствующей области. В этом случае после внесения изменений мы даже можем сохранить предыдущую версию, чтобы, возможно, эти изменения отменить, если нужно. В принципе, мы можем перенести эту схему и на распределенную модель хранения данных. Тогда каждый источник может определить полномочия других пользователей информационной системы и контролировать свой «куст» данных. Недостатком этого подхода является обязательное наличие серверной части, отслеживающей полномочия всех пользователей. Кроме того, свойства этих объектов тогда становятся частью «куста» данных определенного владельца, хотя могут не соответствовать его подходу. В принципе, обозначенные недостатки могут не являться проблемными местами информационной системы.

Рассмотрим случай с отсутствием серверной части, определяющей, какие объекты можно редактировать. Например, чужие «кусты» мы просто получаем при доступе через http-протокол, и там не предусмотрено редактирование.

В этом случае нам все равно требуется внести изменения, однако это надо сделать исключительно в своем контексте редактирования. При этом при визуализации изменения должны корректно показываться, то есть, по крайней мере, в нашей части системы они должны выглядеть именно так, как мы хотим их увидеть.

Итак, чтобы внести изменения в свойства метаобъекта, который находится в удаленном «кусте», предлагается следующий вариант решения. Мы создаем новую сущность у себя в контексте редактирования. Наделяем ее при этом всеми теми же свойствами, что были в удаленном объекте. Теперь этот метаобъект находится в нашем «кусте», и мы можем произвести все необходимые нам изменения. Однако в семантической сети ссылки произ-

водятся все еще на старый объект. В своем контексте редактирования мы, в принципе, можем исправить все ссылки на старый объект на ссылки на новый добавленный объект, однако ссылки на старый объект остаются во всех остальных источниках. Поэтому мы добавляем дополнительное указание для просматривающей системы, что произошла смена идентификатора, и новый объект «заменил» старый. Тогда информационная система должна при получении старого идентификатора автоматически его заменять на новый.

Аналогично мы можем поступать и в случае, когда у нас больше чем 2 источника, и каждый из них областью просмотра имеет все источники, а также может редактировать свойства всех увиденных метаобъектов. В этом случае нам требуется заменить идентификатор столько раз, сколько было совершено этих копирований метаобъектов из одного источника в другой.

При некотором изменении данной схемы мы можем аналогично удалять метаобъекты из чужих «кустов», обозначая в нашем контексте, что этого метаобъекта не существует (для нас).

5. РЕДАКТИРОВАНИЕ ПЕРВИЧНЫХ ОБЪЕКТОВ СОХРАНЕНИЯ

Напомним, что первичные объекты сохранения — это то, что хранится в данном конкретном архиве, музее, библиотеке, т.е. это могут быть фотографии, аудиозаписи, видеодокументы, просто текстовые документы в отсканированном виде и тому подобное. Во-первых, нам надо определиться, в каком случае первичные объекты нуждаются в редактировании, и стоит ли их редактировать вообще. Само действие может быть простейшим, тем не менее, требуется определить, действительно ли редактирование не нанесло ущерб исходным данным.

Простым редактированием могут быть поворот либо вертикальное/горизонтальное отражение документа. Так же это могут быть вырезка белых полей, цветовое улучшение, профессиональное восстановление документа дизайнером и так далее.

Кроме того, для разных интерфейсов нашей информационной системы нам могут потребоваться различные версии одного и того же документа. Например, для их печати нам требуется максимальное качество, для просмотра нам требуется оптимальное соотношение между качеством и временем загрузки. Для веб-интерфейса нам требуются соотношение между качеством, разрешенным владельцем документа для выставления в Интернет,

и размером занимаемой им памяти, а также, как правило, еще «предпросмотровый» вариант этого же документа меньшего размера.

Независимо от того, в каком формате документ попадает в архив, если с ним производить изменения (например, поворот) в архиве, его целесообразно сохранять уже в формате, который исключает потерю данных при сохранении (независимо от того используется компрессия или нет).

Если заменять первичный объект после редактирования, то, возможно, по каким-то параметрам у нас может произойти потеря информации. Поэтому с точки зрения сохранения первичных объектов при более сложных редактированиях, например художественных, лучше добавить дополнительный первичный объект в базу данных, а проблему отождествления решать уже на уровне метаобъектов.

6. РЕЗЕРВНОЕ КОПИРОВАНИЕ И АВТО-ОБНОВЛЕНИЯ

На мировом уровне вопросы резервного копирования решены методологически и технологически во многих аспектах. В данном разделе указаны нюансы применимости этих решений в нашей распределенной модели хранения данных.

Решения должны учитывать следующие особенности. Во-первых, ядром информационной системы в данной модели может служить вовсе не серверная конфигурация компьютера, а также не обязательно наличие источника бесперебойного питания. Во-вторых, возможно отсутствие дополнительной надежной архивной машины (для хранения резервной информации). В-третьих, для увеличения числа машин, являющихся хранителями части информации и входящих в единое информационное поле, одно из требований — минимальная настройка этих машин под данную задачу, т.е. требуется использование наиболее распространенных и встроенных в операционные системы программного обеспечения и интерфейсов программирования приложений (API), например, Microsoft .NET Framework, Java Virtual Machine, Microsoft IIS Server.

Вопросы резервного копирования касаются не только первичных данных и базы метаобъектов, но также и версий программного обеспечения, стилей оформления, структуры данных, различных конфигурационных файлов.

С другой стороны, нам требуется поддержка не только резервного копирования данных, но и публикации новых появившихся первичных объектов или новых версий программного обеспечения.

В плане развития и улучшения фактографической информационной системы мы должны предусмотреть производственную цепочку изменений. Кроме публикации на внешнем сервере (публичный интерфейс), либо на компьютере конечного пользователя нам также требуется возможность тестирования новых версий программного обеспечения. При этом целесообразно использование реальных данных для тестирования. Поэтому требуется несколько серверов (или виртуальных машин), для того чтобы разные данные и разные программные системы не смешивались. Кроме того, необходимо, чтобы данные и системы были как можно более новые, чтобы тестирование проходило в условиях, приближенных к реальности. Для этого должен быть реализован контроль обновления данных и программного обеспечения в имеющихся у нас разных источниках/ресурсах.

При использовании описанных в предыдущих разделах механизмов редактирования первичных объектов и метаобъектов у нас также появляется методологическая возможность не только резервного копирования работающей версии, но и отката системы назад. Для первичных объектов это возможно, так как при их изменении старый объект не удаляется, а все отождествление идет на уровне метаобъектов. Для метаобъектов это возможно при помощи использования описанной реализации концепции распределенного хранения. В определенный момент, когда требуется провести резервное копирование, мы переводим редактируемый контекст в разряд внешних источников. При этом мы все еще можем его дальше редактировать, однако, используя преимущества нашей онтологии и подход, который уже был задан в этой статье, мы не редактируем непосредственно эти данные, они находятся фактически в другом RDF-документе. Вследствие этого мы можем, если нам требуется, совершить откат назад.

Также нас интересует возможность откатов назад и в версиях программного обеспечения. Приемы, которыми этого можно добиться, во многом уже описаны в другой литературе.

7. РЕАЛИЗАЦИЯ

Работа велась в рамках проекта «Электронный фотоархив Сибирского отделения Российской академии наук». В проекте требовалось создать информационную систему для сохранения документов и добавления к ним метаданных. С целью привлечения к проекту дополнительных информационных ресурсов необходима возможность последовательной интеграции данных с установлением распределенных механизмов хранения и доступа.

Подходы, рассмотренные выше, были реализованы в виде модулей, взаимодействие которых определяет информационную систему. Помимо интерфейсов для ввода, редактирования и внутреннего поиска информации, существует и «легкий» публичный интерфейс, доступный по адресу: http://soran1957.ru/

СПИСОК ЛИТЕРАТУРЫ

1. Марчук А.Г. Распределенные электронные архивы, библиотеки и базы данных // Новосибирск, 2004. — 25 с. — (Препр. / ИСИ СО РАН; № 122).
2. Resource Description Framework (RDF) . — http://www.w3.org/RDF/
3. Sesame, open source RDF framework. — http://openRDF.org/
4. Jena — A Semantic Web Framework for Java. — http://jena.sourceforge.net/
5. Berners-Lee Tim, Hendler James, Lassila Ora, The Semantic Web // Scientific American. — 2001. — Vol. 284(5). — P. 34–43.
6. Марчук П.А. Особенности интеграции данных из разных источников // Технологии Microsoft в теории и практики программирования / Конф.-конкурс работ студентов, аспирантов и молодых ученых. Тез. докл. — Новосибирск, 2007 — С. 129–131.
7. The Dublin Core Metadata Initiative (DCMI) . — http://dublincore.org/
8. Web Ontology Language (OWL) . — http://www.w3.org/2004/OWL/
9. SPARQL Query Language For RDF. — http://www.w3.org/TR/rdf-sparql-query/

Г. П. Несговорова

ОРГАНИЗАЦИЯ ИСТОРИКО-КУЛЬТУРНОГО ПРОСТРАНСТВА В ИНТЕРНЕТЕ С ИСПОЛЬЗОВАНИЕМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ВВЕДЕНИЕ

В современном мире, где непрерывно совершенствуются информационные и коммуникационные технологии, происходит их стремительное проникновение в различные сферы человеческой деятельности. Одной из таких сфер является культура, одна из важнейших составляющих деятельности человека с древнейших времен. Сохранение культурного наследия каждого народа и в целом мировых культурных ценностей — это то, без чего не может быть дальнейшего развития человечества. Люди давно поняли это и всегда старались сохранять историко-культурные объекты, даже в самые трудные времена в жизни разных стран и народов.

В настоящее время в деле сохранения культурного наследия во всем мире применяются информационные технологии с выходом в мировое информационное пространство, в частности, во всемирную сеть Интернет. В связи с этим музеи и прочие заведения культуры внедряют новые информационные, цифровые и коммуникационные технологии в свою деятельность, что значительно улучшает их работу по учету, сохранности, поиску и популяризации объектов культурного наследия.

Информационные и коммуникационные технологии объединяют людей и мировое культурное наследие. Это то, что интересует всех и принадлежит всем: интерес к историческим событиям, к культуре разных стран и эпох — все это вечные ценности. Поэтому необходимо интенсивно переводить в цифровой вид информацию о мировом культурном наследии для ее сохранности и простоты доступа к ней любого пользователя в любой точке земного шара.

Наступает электронный век культуры, в который происходит формирование оптимальных условий, обеспечивающих эффективное применение и развитие информационных и телекоммуникационных технологий в культуре и сохранение на их основе культурного наследия, выработка единой концепции развития информатизации разных областей культуры. Современное информационное общество выдвигает новые требования к формированию информационных ресурсов культурной сферы, рассматривая культурное наследие как один из элементов территориальной системы.

Поскольку разрабатываемый в лаборатории конструирования и оптимизации программ (КОП) Института систем информатики СО РАН имени А.П. Ершова виртуальный музей истории информатики в Сибири [1] является малой частицей мирового культурного пространства, поэтому, создавая его, хотелось бы формально уточнить и сформулировать, с какими понятиями и объектами мы работаем, а именно: что такое культурное наследие в целом, что является объектом культурного наследия, какие существуют технологии сохранения объектов культурного наследия и стандарты их описания. В статье приводятся также примеры российской и мировой сети по сохранению и популяризации культурного наследия.

1. ОБЪЕКТ КУЛЬТУРНОГО НАСЛЕДИЯ

Слово «культура» есть во всех языках мира. Оно означает «возделывание, изменение, улучшение», производимое человеком в процессе целесообразной деятельности. Весь окружающий нас мир — это мир культуры. Он включает в себя орудия труда, средства производства и транспорт, технические изобретения и научные открытия, язык и письменность, произведения искусства, нормы морали и пр.

Культурное наследие — это совокупность всех материальных и духовных культурных достижений общества, его исторический опыт, сохраняющийся в арсенале общественной памяти, иными словами, совокупность объектов и явлений материального и духовного творчества людей, имеющая историческое значение и подлежащая сохранению. Культурное наследие составляют достижения различной давности, переходящие к новым поколениям в новые эпохи от предыдущих.

Культурное наследие России (согласно законодательству РФ) включает в себя памятники, произведения архитектуры, монументальной скульптуры и живописи, элементы или структуры археологического характера, исторические надписи, пещерные жилища и группы их элементов, ансамбли (группы изолированных или объединенных строений), их единство или связь с пейзажем, ландшафтно-исторические комплексы и пр., которые имеют выдающуюся универсальную ценность с точки зрения истории, искусства, науки.

Всемирное культурное наследие — это выдающиеся природные и культурные ценности разных народов и стран, составляющие достояние всего человечества.

Прежде чем говорить о сохранении культурного наследия в информационном пространстве, рассмотрим, что относят к объектам культурного наследия (ОКН).

Как правило, объекты культурного наследия подразделяются на *материальные* и *духовные*.

Материальные объекты — это те объекты, которые имеют реальное вещественное воплощение в реальном мире, их можно увидеть, потрогать руками. Материальные ОКН, в свою очередь, подразделяются на *недвижимые* и *движимые*. К *недвижимым* относятся разного рода памятники, посвященные выдающимся людям и событиям, архитектурные и археологические сооружения, а также исторические и культурные ландшафты, включая поля сражений и старинные усадьбы, являющиеся одновременно и архитектурными и ландшафтными ОКН. Сюда же включают зоны природных заповедников и заказников. *Движимые* объекты культурного наследия — это предметы музейных фондов, архивов, библиотек, картинных галерей и пр. независимо от их форм собственности.

К материальным ОКН, как правило, относят также произведения музыкального искусства предшествующих эпох — это различные музыкальные произведения: песни, романсы, оперы, симфонии, оратории, оперетты и т.д. Сюда же входят театральное и изобразительное, танцевальное и балетное искусства, киноискусство разных жанров, литература, поэзия и пр. Объекты этого культурного наследия имеют свое материальное воплощение в виде записей музыкальных произведений на нотных листах и разного рода аудионосителях, в виде текстов драматических произведений на бумаге и реальных декораций на сцене, картин, висящих в музеях и картинных галереях, кино- и видеоархивов на различных видеоносителях, книг и журналов. Их также относят к движимым объектам материального культурного наследия.

Духовное (нематериальное) культурное наследие [2] объединяет обычаи, знания и навыки, а также связанные с ними разного рода инструменты, предметы, артефакты и культурные пространства, признанные группами или сообществами людей в качестве части их культурного наследия. Такое духовное наследие, передаваемое от поколения к поколению, постоянно воссоздается сообществами, формируя у них чувство самобытности и преемственности. Это сокровищница духа и души народа.

Формами проявления духовного культурного наследия являются:

- устные традиции и формы их выражения, включая язык в качестве носителя наследия;
- исполнительские искусства;

- обычаи, обряды, празднества;
- знания и обычаи, относящиеся к природе и вселенной;
- знания и навыки, связанные с традиционными ремеслами;
- религиозные и культовые праздники и обряды.

Духовно-культурное наследие используется сообществом в духовно-нравственном воспитании новых поколений.

В целях сохранения культурного наследия в 1996 г. в России был принят федеральный закон «О музейном фонде Российской Федерации и музеях в Российской Федерации». В нем даются следующие определения основных понятий и терминов, связанных с музейным фондом и музеями, с которыми стоит ознакомиться.

Культурные ценности — предметы религиозного или светского характера, имеющие значение для истории и культуры.

Музейный предмет — культурная ценность, качество или особые признаки которой делают необходимым для общества ее сохранение, изучение и публичное представление.

Музейная коллекция — совокупность культурных ценностей, которые приобретают свойства музейного предмета, только будучи соединенными вместе в силу характера своего происхождения, либо видового родства, либо по иным признакам.

Музейный фонд — совокупность постоянно находящихся на территории Российской Федерации музейных предметов и музейных коллекций, гражданский оборот которых допускается только с соблюдением ограничений, установленных Федеральным законом.

Музей — некоммерческое учреждение культуры, созданное собственником для хранения, изучения и публичного представления музейных предметов и музейных коллекций.

Хранение — один из основных видов деятельности музея, предполагающий создание материальных и юридических условий, при которых обеспечивается сохранность музейного предмета и музейной коллекции.

Публикация — одна из основных форм деятельности музея, предполагающая все виды представления обществу музейных предметов и музейных коллекций путем публичного показа.

2. ТЕХНОЛОГИИ СОХРАНЕНИЯ ОБЪЕКТОВ КУЛЬТУРНОГО НАСЛЕДИЯ

В настоящее время технологии сохранения объектов культурного наследия постоянно совершенствуются благодаря развитию новых методик,

материалов и появлению новых приемов работы с ними. Технологии сохранения объектов культурного наследия подразделяются на обычные - классические, существующие веками, и современные — информационные технологии, получившие свое развитие в связи с появлением и внедрением в сферу культуры компьютеров.

К классическим технологиям сохранения ОКН относятся

1) восстановление (реконструкция) разрушенных и сохранение (консервация) существующих объектов культурного наследия;
2) реставрация (произведений живописи, икон, фресок, фасадов зданий, фото, книг, предметов интерьера и пр.);
3) разработка и использование новых материалов в перечисленных выше работах;
4) создание и реализация методик по сохранению объектов культурного наследия, включая также историко-природные ландшафты, историко-литературные и природные музеи-заповедники (например, усадьбы писателей, поэтов, художников и других выдающихся деятелей).

Обычные технологии — реставрация, консервация, реконструкция — имеют свои новейшие разработки, ноу-хау и достижения. Существуют лаборатории по реконструкции и реставрации памятников архитектуры и археологии, мастерские по реставрации фасадов и интерьеров, картин, исторических документов, фото, отдельных музейных предметов, в которых работают специалисты в области восстановления и сохранения различных типов ландшафтов.

Что касается электронных или информационных технологий сохранения объектов культурного наследия, то они начали развиваться с конца прошлого века и стремительно продолжают совершенствоваться в настоящее время. Для создания базы данных по культурному наследию требуются стандартизованная терминология описания музейного предмета (или любого другого объекта культурного наследия), стандартизованная технология по созданию и постоянному пополнению государственного тезауруса по музейной терминологии, а также стандартизированная технология конверторов для формирования единого музейного пространства.

Для сохранения, представления и использования объектов культурного наследия используются стандартизованные методы и средства информационно-коммуникационных технологий, а именно:

1) создание массивов баз данных;
2) разработка стандартизированных информационно-поисковых систем в области культурного наследия;

3) применение стандартных электронных словарей и справочников;
4) использование сети коммуникаций, в частности, всемирной сети Интернет;
5) создание музейных сайтов, виртуальных музеев и пр.;
6) сканирование и оцифровка различных музейных, изобразительных, текстовых, архивных, фото-, аудио-, видео- и прочих экспонатов;
7) развитие методологической базы по оцифровке ОКН, совершенствование собственно компьютерных технологий, программного обеспечения и цифрового оборудования.

В проект по сохранению культурного наследия с помощью информационных технологий включены музеи, библиотеки, архивы, картинные галереи. Примером использования технологии оцифровки произведений искусств может служить применение сканера «ЭПОС». Это современное оборудование, позволяющее оцифровывать музейные оригиналы без сильного теплового и светового воздействия и обеспечивающее тем самым их сохранность. Цифровые копии архивных и библиотечных единиц — это новый способ воспроизведения и сохранения ОКН и новая веха в обеспечении информационной безопасности архивов, книг и других экспонатов в электронном виде в сравнении с их обычным хранением на стеллажах. Цифровые копии занимают мало места, менее подвержены колебаниям температуры и влажности, их можно хранить в разных «корзинах» на случай стихийных бедствий. Использование информационных технологий, таким образом, эффективно помогает в популяризации и сохранении культурного наследия, а основа успеха в такого рода деятельности — это творческий союз специалистов в области информационных технологий и представителей сферы культуры.

3. СТАНДАРТ ОПИСАНИЯ ОБЪЕКТОВ КУЛЬТУРНОГО НАСЛЕДИЯ

Проблема создания стандартов описания объектов культурного наследия стоит перед всеми, кто занимается их представлением в сети Интернет. Решение этой проблемы заключается в разработке единого для всех музеев, независимо от их профиля, стандарта краткого описания или, другими словами, — этикетки объекта культурного наследия. Это необходимо прежде всего для создания Российской сети культурного наследия и представления в этой сети данных о коллекциях российских учреждений культуры.

На основе подробного анализа российского и зарубежного опыта была предложена модель единого стандарта краткого описания (этикетки) разнородных объектов культурного наследия: музейных и галерейных коллекций, памятников архитектуры и археологии, объектов культурно-исторического и природного ландшафтов. Таким образом, в течение нескольких лет различными специалистами был разработан объединенный стандарт — рекомендации по созданию и внедрению единой этикетки - и предложена структура гипотетической общей этикетки для объектов культурного наследия.

В стандарт краткого описания объектов культурного наследия (БД этикеток) входят:

1) название,
2) местоположение,
3) типология,
4) период,
5) датировка,

6) персоналии,
7) автор,
8) идентификационный номер,
9) размеры,
10) краткое описание,
11) форма собственности,
12) категория охраны.

При этом 1–5 поля являются обязательными общими полями, а 6–12 и т.д. — дополнительными общими полями и используются по усмотрению создателей описания конкретного объекта культуры. Такого рода этикетки используются для представления оцифрованных объектов культурного наследия в электронных базах данных и разного рода музеях в сети Интернет.

4. РОССИЙСКАЯ СЕТЬ КУЛЬТУРНОГО НАСЛЕДИЯ В ИНТЕРНЕТЕ

Наблюдающееся в последние годы резкое ускорение развития и использования информационных и коммуникационных технологий послужило началом всемирного процесса перехода от «индустриального» к «информационному» обществу. В основе этих преобразований лежат технологические достижения, к которым относятся: цифровая обработка различных

видов информации — текста, чисел, звука, изображения — и их интеграция в единый продукт, называемый «мультимедиа», техника цифрового уплотнения и переключения, рост мощности компьютеров и взрывной рост компьютерных сетей — и крупнейшей из них — Интернета, который охватывает миллионы пользователей во всем мире.

Российская сеть культурного наследия (РСКН) — это некоммерческая организация, основанная в 1996 г. и учрежденная Министерством культуры России. РСКН осуществляет ряд информационных проектов, направленных на продвижение Российского культурного наследия в сторону мирового сообщества, формирование открытой коммуникационной среды для специалистов в области изучения культурного наследия. Она выступает в качестве координационного, консультативного и информационного центра, обеспечивает менеджмент и технологическое сопровождение информационных проектов, отвечает за взаимодействие с зарубежными партнерами. Используя информационные технологии, РСКН ведет активную пропаганду знаний о культурном наследии России под девизом: «Сохраняйте прошлое и настоящее для будущего».

Примерами основных проектов Российской сети культурного наследия являются:

1) проект Европейской комиссии «Cultivate-Russia» — сетевой, инфраструктурный проект (11 партнеров из 6 европейских стран), направленный на пропаганду сотрудничества организаций России и Европы в области культуры (`http://www.Cultivate.ru`);

2) Интернет-портал «Музеи России» — основной музейный сервер России. Основан в 1996 г., доступ ко всей информации бесплатный. Содержит исчерпывающую информацию о более чем 3000 музеев и галерей. Размещает общероссийские музейные афиши и анонсы событий культурной жизни. Информация обновляется ежедневно. Ежедневно посещается 70 000 российских и зарубежных пользователей, более 300 000 обращений каждый день (`http://www.Museum.ru`);

3) Интернет-портал «Культура России» — официальный сервер Министерства культуры Российской Федерации. Рассчитан на массового пользователя, представляет различные срезы информации по культуре на протяжении всего времени существования России
(`http://www.RussianCulture.ru`);

4) Интернет-портал «Библиотеки России». Представляет централизованные библиоресурсы России (`http://www.Libs.mincult.ru`);

5) информационное агентство «Культурное наследие» и информационная служба музеев России. Освещает новости музеев и галерей, является

общероссийской музейной афишей и корреспондентской сетью, размещает обзоры и аннотации (http://www.Museum.ru.News/);

6) Всероссийский реестр музеев и галерей является системой связанных баз данных, обеспечивает мониторинг и статистику. Выработаны стандарты описания музейной организации, предметов фондов, типология музеев. Разработано специальное программное обеспечение, обслуживающее массивы данных (http://VRM.museum.ru);

7) сервер музейных профессионалов. Цель этого проекта — превратить Интернет в полноценный рабочий инструмент музейного специалиста. Ориентирован на профобщение и оперативное предоставление информации о музейной деятельности: новости, отклики, рецензии, координаты близких к музеям организаций, список фирм, реализующих музейное оборудование, ссылки на профессиональные сайты, информация о конференциях, грантах, конкурсах и т.д. (http://www.Museum.ru/Prof);

8) Интернет-сервер, представляющий зоопарки России и ее ландшафтно-природное наследие (http://www.Zoo.ru);

9) сайт Государственного музея изобразительных искусств им. А.С.Пушкина, одного из самых старейших музеев России (http://www.Museum.ru/gmii);

10) сайт Российской государственной библиотеки, одной из крупнейших библиотек мира (http://www.RSL.ru);

11) агентство Культурной Информации — совместный проект с газетой «Культура» (http://www.AKI-ros.ru);

12) республиканский портал «Музеи Татарстана» — первый интеграционный музейный Интернет-проект (http://www.Tatar.museum.ru).

В настоящее время происходят положительные сдвиги в развитии и использовании информационно-коммуникационных технологий в сфере культуры регионов России на базе региональных программ, а также в странах СНГ, примером чего является Казахстан (http://www.chsp.rz). Этот сайт опубликован на казахском, русском и английском языках и представляет материалы по казахскому культурному наследию, а также официальные документы Государственной Программы «Культурное наследие Казахстана».

5. ИНТЕГРАЦИЯ РОССИЙСКОГО КУЛЬТУРНОГО НАСЛЕДИЯ В МИРОВОЕ КУЛЬТУРНОЕ ПРОСТРАНСТВО

Надо отметить, что Российский культурный сектор Интернета отстает от уровня развития мировых информационных систем аналогичной тематики. В объединенной Европе существует проект по Европейскому культурному наследию («Open Heritage») — «Открытое наследие: создавая европейскую экономику культуры». Он создается в рамках программы Европейского Совета «Технологии информационного общества» и его целью является создание самоокупаемой и устойчивой модели существования культуры в современном мире.

Применение информационных и коммуникационных технологий создает перспективные возможности для использования богатейших культурных ресурсов Европы. Они открывают новые пути для сохранения, накопления, описания и открытия содержания многочисленных архивов, библиотек, музеев, реконструкции и визуализации артефактов и археологических сайтов. Исследования стран Евросоюза в области цифровой культуры — это часть упомянутой выше программы «Технологии информационного общества». Задачи этой программы:

1) сделать легким и доступным для людей поиск, понимание и освоение их культурного наследия посредством виртуального посещения цифровых библиотек и визитов в прошлое — осмотр электронных музеев;

2) сохранить аудио-, видео- и кинонаследие XX века;

3) создавать и охранять сегодняшнюю «цифровую» культуру для будущего.

В сфере сохранения мирового культурного наследия в целом необходимо сотрудничество по развитию и использованию информационно-коммуникационных технологий разных стран мира. Важную роль в этом процессе играют Бюро ЮНЕСКО в г. Москва, Российский комитет программы ЮНЕСКО «Информация для всех» и европейские проекты серии MINERVA [3], куда входит и Россия.

Одним из текущих проектов является проект BRICKS — это интегрированный проект, нацеленный на создание организационных и технологических основ для Цифровой Библиотеки на уровне European Digital Memory. В нем предлагаются функциональные возможности для нового поколения цифровых библиотек, включающие в себя «Цифровые архивы», «Цифровые музеи» и другие системы цифровой памяти.

Таким образом, сейчас во всем мире разрабатывается и переходит в фазу реализации идея информатизации мирового культурного наследия. Ин-

теграция информационных ресурсов по всемирному культурному наследию должна происходить в двух плоскостях: вертикальном — национальное культурное наследие — и горизонтальном — межнациональное культурное наследие разных стран и народов. Одна из основных задач в этой области для России — с одной стороны, сделать российское культурное наследие доступным всему миру и, с другой, предоставить российским пользователям мировое культурное пространство во всем его многообразии, обеспечив к нему простой и быстрый доступ.

ЗАКЛЮЧЕНИЕ

В заключение хочется отметить, что в современных условиях информационная инфраструктура как совокупность информационных ресурсов и программно-аппаратных средств вычислительной и телекоммуникационной техники, информационных технологий и вычислительных сетей является одним из важнейших компонентов любого вида деятельности, в том числе и культурной деятельности человека. Технологии информационного общества занимают ключевую роль в сохранении историко-культурного и научного наследия для будущих поколений и в предоставлении широкому кругу пользователей доступа к ним.

В настоящее время растет число проектов в области сохранения и расширения доступа к объектам культурного наследия, направленных на эффективное использование и популяризацию культурного наследия как части информационно-просветительской работы.

СПИСОК ЛИТЕРАТУРЫ

1. Касьянов В.Н., Несговорова Г.П., Волянская Т.А. Виртуальный музей истории информатики в Сибири // Современные проблемы конструирования программ. — Новосибирск, 2002. — С. 169–181.
2. Нематериальное культурное наследие под охраной ЮНЕСКО (материалы вебсайта). — http://un.ru/bulletin/
3. Несговорова Г.П. Современные информационно-коммуникационные и цифровые технологии в сохранении культурного и научного наследия и развитии музейного дела // Проблемы интеллектуализации и качества систем информатики. — Новосибирск, 2006. — С. 153–161.

К. А. Пыжов

ВНУТРЕННИЕ ПРЕДСТАВЛЕНИЯ СРЕДНЕГО УРОВНЯ ДЛЯ КОМПИЛЯТОРОВ ЯЗЫКА SISAL*

ВВЕДЕНИЕ

Функциональные языки и языки однократного присваивания имеют семантику, базирующуюся на понятии «значение», а не «ячейка памяти», и это дает им ряд преимуществ. В частности, такой подход позволяет трансляторам функциональных языков осуществлять эффективное распараллеливание вычислений. Отсутствие побочных эффектов делает анализ зависимостей по данным и возможностей параллелизма гораздо более легкой задачей, чем в случае с императивными языками. В частности, отпадает необходимость трудоемкого анализа зависимостей по данным для применения тех или иных оптимизирующих преобразований, а также для распараллеливания и векторизации. Во внутреннем представлении потоковых языков зависимости по данным представлены в явном виде. К слову, предлагаются различные механизмы, позволяющие императивным языкам использовать преимущества концепции однократного присваивания. Например, в последнее время в компиляторах для императивных языков широкое распространение получила SSA-форма [1, 2], которая вносит в представление программы положительные стороны языков однократного присваивания. SSA-форма позволяет эффективно удалять избыточные зависимости по данным и, тем самым, расширяет и упрощает применение оптимизирующих преобразований.

К функциональным языкам однократного присваивания, свободным от побочных эффектов, относится Sisal [8]. Sisal — довольно известный потоковый язык, предназначенный для параллельных вычислений. От языка VAL отличается поддержкой ошибочных значений, допустимостью рекурсии, потоковым типом данных. Кроме того, Sisal имеет наглядный синтаксис, сходный с языком Pascal. Все эти особенности делают язык Sisal удобным для записи больших программ, содержащих сложные математические расчеты. Однако с точки зрения компилятора

*Работа частично поддержана Российским фондом фундаментальных исследований (грант РФФИ № 07-07-12050).

концепция однократного присваивания и семантика, базирующаяся на значениях, создают специфические трудности, связанные прежде всего с необходимостью использования больших объемов динамической памяти и управлением параллельно исполняемыми задачами.

Нужно отметить два основных источника неэффективности. Первый — это необходимость создания большого количества динамических объектов. Для эффективного использования памяти необходимо организовать хорошее управление этими объектами во время исполнения программы. Второй источник неэффективности возникает как раз из-за отсутствия побочных эффектов: изменение элементов структурного объекта (такого как массив, например) семантически предполагает создание новой копии всего объекта. Такое копирование может быть достаточно «дорогим» и значительно влиять на эффективность кода. Однако существует возможность во многих случаях избегать таких копирований. Одним из подходов к решению этой задачи является создание промежуточного представления среднего уровня, которое наследует потоковую структуру первого представления, но при этом содержит информацию о размещении объектов в памяти [6]. Такая информация позволяет организовать оптимизацию использования динамических объектов. Во многих случаях удается избежать необходимости копирования всего объекта, ограничиваясь модификацией какой-либо его части. Понятно, что такого рода преобразования требуют выполнения определенных условий использования объектов и выполнения соответствующего анализа программы.

В данной статье представлено краткое описание некоторых проблем, возникающих при трансляции функциональных программ и методов их решения.

В первой и второй частях статьи содержится обзор различных методов, применяющихся для оптимизации работы с памятью и управления многозадачностью в компиляторах функциональных языков.

В третьей части дано краткое описание внутренних представлений, разработанных для нового компилятора языка Sisal 3.1, разрабатываемого в Лаборатории конструирования и оптимизации программ ИСИ СО РАН.

1. ТЕХНОЛОГИИ ОПТИМИЗАЦИИ РАБОТЫ С ПАМЯТЬЮ

В Ливерморской лаборатории в рамках работы над созданием оптимизирующего транслятора языка Sisal 1.0 было разработано проме-

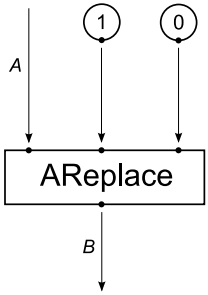


Рис. 1. Операция AReplace

жуточное представление IF2 (наряду с представлением верхнего уровня IF1), расширяющее представление IF1 операциями распределения и управления памятью для объектов [5]. Если представление IF1 не несет никакой информации о размещении частей объектов в памяти, то IF2 делает некоторые предположения: например, требование размещения элементов одномерного массива в последовательных ячейках памяти. Операции со скалярными значениями остаются неизменными.

Пример 1. На рис. 1 показано потоковое представление операции AReplace языка Sisal, выполняющей замену элемента массива (Первый элемент массива заменяется на 0). Операция задается следующим выражением языка Sisal:

```
B := A[1:0].
```

В семантике представления IF1 операция AReplace создает новую копию массива A с измененным первым элементом. Соответственно, необходимо создание нового динамического объекта для новой копии массива. Однако если AReplace является последним использованием массива A, мы можем записать новое значение в область памяти, используемую для первого элемента этого массива, и использовать память, отведенную для A, для размещения массива B. В графе IF2 дуга, представляющая массив A, будет помечена специальным свойством, указывающим, что к этому массиву возможно обращение «по ссылке», т.е. обращение не только к значению, но и к соответствующей области памяти. При этом для новой копии массива будет использован тот же самый буфер динамической памяти.

Кроме того, в представление IF2 добавляются искусственные дуги

зависимостей. Дело в том, что в некоторых случаях бывает удобно задержать выполнение оператора над большой структурой данных до тех пор, пока все остальные ссылки на эту структуру не будут вычислены. Такая задержка позволит операции обращаться к данным по ссылке, избегая таким образом копирования. В качестве примера рассмотрим представление для фрагмента программы на рис. 2. Если задержать исполнение оператора AElement до завершения операции ASize, то AElement сможет работать с массивом A по ссылке, без создания новой копии. Для обозначения этой задержки в граф вводится дуга искусственной потоковой зависимости, которая проводится от вершины ASize к вершине AElement.

```
function length(N:integer; L:integer  returns integer, integer)
   let
      A:= for I in 1, N
             returns array of I
          end for
   in
      size(A),
      A[L]
   end let
end function
```

В Стенфордском университете был разработан back-end транслятор для IF1, в котором основное внимание уделялось динамическому распределению памяти для массивов и процедуре освобождения динамической памяти [4]. Механизм освобождения памяти был основан на подсчете ссылок (reference counting). Разработанный менеджер памяти имел такие механизмы, как передача параметров-структур по указателям и разделение доступа к структурам. Для упрощения задачи динамического подсчета ссылок на объекты была реализована двухпроходная схема кодогенерации: на первом проходе по графу IF1 происходила генерация информации о последних использованиях структур (само представление IF1 никак не определяет порядок исполнения своих элементов), на втором проходе осуществлялась непосредственно кодогенерация.

В 1989 г. в университете Колорадо совместно с Ливерморской лабораторией был разработан компилятор OSC (Optimizing Sisal Compiler) [8]. Процесс трансляции состоял из следующих фаз:

1) font-end трансляция в представление IF1;
2) построение единого IF1 графа программы;
3) build-in-place анализ и трансляция в IF2;

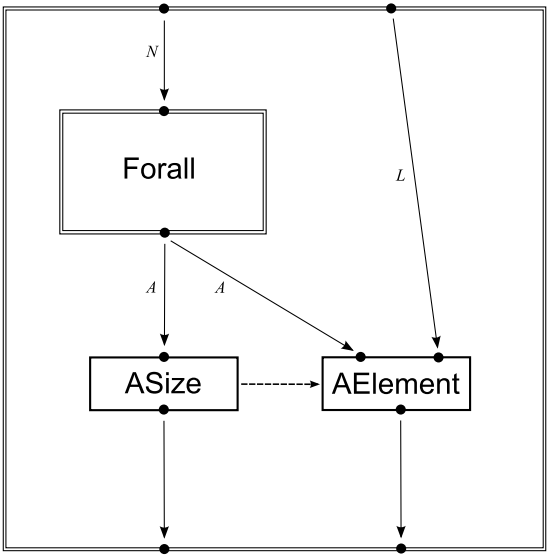


Рис. 2. Искусственные дуги зависимости в IF2

4) update-in-place анализ;

5) кодогенерация.

После построения единого графа программы на представлении IF1 производятся такие оптимизации, как межпроцедурная протяжка значений, подстановка функций, вынос инвариантного кода, удаление общих подвыражений, свертка константных вычислений, удаление мертвого кода.

Задачей build-in-place анализа является решение проблемы конструирования объектов путем предварительного распределения буферов памяти для массивов вне зависимости от того, является ли размер массива известным во время компиляции. Результатом build-in-place анализа является трансляция программы в представление IF2.

Затем программа в представлении IF2 подвергается update-in-place анализу для решения проблемы модификации объектов. Выявляются операции преобразования, которые могут быть выполнены локально. Анализ включает три фазы. Сначала в графы IF2 добавляются специальные вершины, помечающие операторы, дублирующие объекты. Целью оставшихся шагов является удаление ненужных дублирований.

Первая фаза включает также аннотирование каждой дуги, несущей агрегатный тип, значением счетчика ссылок в худшем случае (это возможно на стадии компиляции, поскольку все агрегатные объекты являются ациклическими). Вторая фаза переставляет, где это возможно, вершины в каждом графе, вводя при этом искусственные дуги зависимости. На этом шаге устраняются также счетчики ссылок, оказавшиеся ненужными в результате перестановки вершин. Наконец, заключительная фаза удаляет все избыточные операции дублирования, введенные на первом шаге.

В качестве кодогенератора в компиляторе OSC использовался транслятор оптимизированного представления IF2 в эквивалентную программу на языке C.

2. РАСШИРЕНИЕ ЯЗЫКА SISAL ДЛЯ УПРАВЛЕНИЯ МНОГОЗАДАЧНОСТЬЮ

Практика показывает, что SISAL-программы содержат значительный потенциальный параллелизм. Проблема заключается в преобразовании этого внутреннего параллелизма в множество эффективных задач для многопроцессорной системы. Во-первых, важно выбрать правильную стратегию выделения задач и управления ими во время исполнения программы. Во-вторых, следует производить оптимизацию последовательного кода задач, которая позволит SISAL-коду быть сравнимым по производительности с последовательными программами на таких языках, как C и Fortran.

Но трансляция потока вычислений для исполнения на традиционных «фон-неймановских» архитектурах вызывает трудности. Концепция однократного присваивания и отсутствия побочных эффектов кардинально отличается от концепции последовательного исполнения традиционных архитектур. Осознание необходимости эффективной трансляции функциональных языков для исполнения на таких архитектурах подтолкнуло к идее комбинирования двух концепций (кстати, ставилась и обратная задача — трансляция последовательных языков на потоковые архитектуры. Но эта идея широкого распространения не получила).

SISAL допускает различные модели исполнения. Например, возможны потоковая, макропотоковая, векторная, модель систолических массивов. С точки зрения компилятора конструкциями, вводящими параллелизм, являются параллельные циклы и вызовы функций. Однако при этом не предполагается никаких возможностей для программиста явно

задавать распараллеливание и выделение задач в программе. Большинство компиляторов потоковых языков выделяют независимо исполняемые блоки кода, исходя из итеративных конструкций и вызовов функций, содержащихся в программе. Управление задачами и их синхронизация являются исключительно задачами компилятора. Императивные языки, напротив, предоставляют программисту широкий набор средств для явного описания параллельных частей программы (например, широко распространенная технология OpenMP). Как известно, в потоковых языках распараллеливание базируется на выявлении независимых вычислений (вершин потокового графа). Любые конструкции, не связанные между собой потоком данных, могут быть исполнены параллельно. Если же они связаны, выполнение зависимой конструкции возможно лишь тогда, когда будут вычислены все конструкции, поставляющие данные этой вершине. Исходя их этого, строится стратегия выделения задач и управления ими.

Было предложено расширение языка Sisal [3], позволяющее пользователю задавать процессы в Sisal-программе в явном виде. Явное определение процесса может находиться в любом месте Sisal-программы и включать любые функции и выражения. Выглядит определение процесса следующим образом:

```
process <process-name>
    <Sisal statement>;
    <Sisal statement>;
    ...
    ...
    function f1(...)
    ...
    ...
end process
```

Для управления процессами предусмотрены директивы fork и join. Конструкция fork выглядит так:

```
fork(P_1, P_2, ... P_n);
```

где P_i — имя процесса. Директива fork разрешает разделение n процессов и выполнение их на разных процессорах, если это позволяют ресурсы. Конструкция join имеет вид:

```
join(P_1, P_2, ... P_m);
```

Конструкция join вызывает принудительную синхронизацию процессов: уже завершившиеся процессы будут остановлены до завершения всех процессов из списка $P_1, P_2, ...P_m$.

Компилятор помечает процессы, участвующие в директиве fork, для

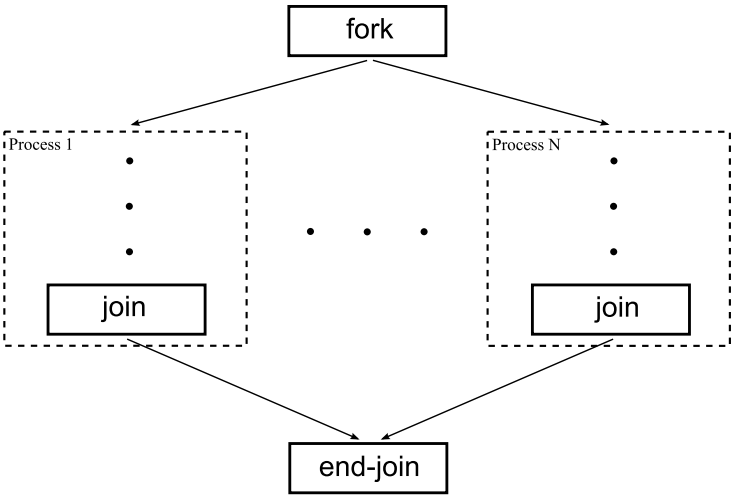


Рис. 3. Синхронизация процессов

последующего параллельного исполнения. В конец каждого процесса из директивы join добавляется специальная вершина, обозначающая, что процесс участвует в синхронизации. Выходы всех таких вершин для всех процессов, обозначенных директивой join, подаются на вход дополнительной «end-join» вершине. Присутствие этой вершины гарантирует одновременное завершение этих процессов (рис. 3).

3. ПРОМЕЖУТОЧНЫЕ ПРЕДСТАВЛЕНИЯ ДЛЯ КОМПИЛЯТОРА ЯЗЫКА SISAL 3.1

В Лаборатории конструирования и оптимизации программ ИСИ СО РАН разрабатывается компилятор, реализующий язык Sisal 3.1. Новая версия языка имеет несколько нововведений, призванных сделать язык более наглядным, гибким и удобным для использования на практике [10]. Однако дело осложняется отсутствием законченной рабочей версии компилятора, а также достаточно широкой тестовой базы. Понятно, что эти проблемы взаимосвязаны.

Первое внутреннее представление (представление front-end транслятора) IR1 является ориентированным ациклическим иерархическим графом потока данных в программе [7]. Каждой дуге приписан тип

значения, которое она несет. Вершины обозначают операции над своими аргументами (входами), результаты которых находятся на выходах вершины. Структурированность вычислений отражается иерархичностью графов IR1. Представление IR1 является полностью машинно-независимым.

Следующее промежуточное представление IR2 представляет собой потоковый граф, подобный IR1, с некоторыми дополнительными конструкциями. В частности, каждой дуге графа приписывается переменная. Дугам, идущим из одного порта, приписывается одна и та же переменная. С использованием этой дополнительной информации осуществляется анализ представления IR2 с целью оптимизации динамической памяти.

Первым этапом построения IR2 является конвертация потокового графа IR1 в аналогичный граф IR2. С точки зрения структуры граф IR2, полученный сразу после трансляции IR1 → IR2, является изоморфным графу IR1, поэтому собственно создание вершин и дуг представления IR2 является тривиальным.

На следующем шаге для дуг, выходящих из каждого порта, создается уникальная переменная, описывающая некоторую область памяти (динамическую или статическую, в зависимости от типа переменной). При этом сохраняется семантика однократного присваивания. Переменные наследуют типы языка Sisal, приписанные к соответствующим дугам графа IR1. Кроме того, каждая переменная описывается такими машинно-ориентированными характеристиками, как двоичный тип, размер и т. д. Таким образом, представление IR2 является машинно-зависимым.

Следующей фазой является оптимизация IR2, которая выполняет оптимизацию распределения переменных агрегатных типов (тем самым удаляя избыточные копирования), а также выносит инвариантные вычисления из графов циклов.

Далее следует трансляция IR2 в IR3. Для каждой вершины представления IR2 вычисляется приоритет ее исполнения (на основе потока данных). Затем для вершин строятся последовательности операторов IR3, которые располагаются согласно приоритетам исполнения вершин IR2.

Представление IR3 является высокоуровневым императивным представлением. В процессе трансляции IR2 в IR3 из конструкций IR2 выделяются простые операторы. Переменные IR2 наследуются.

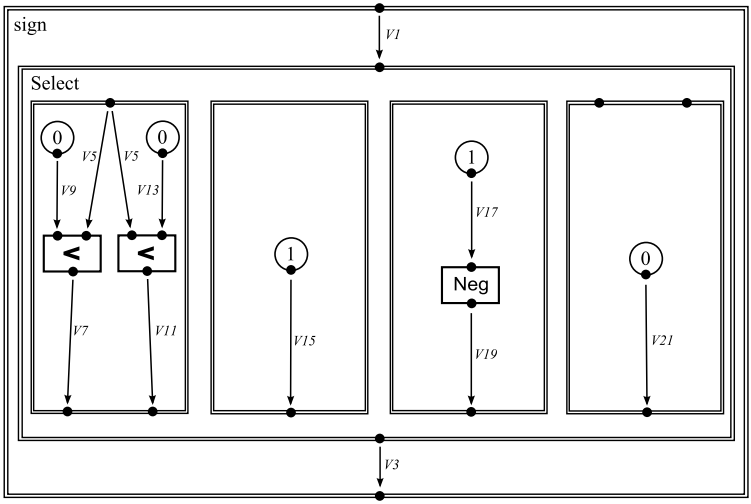


Рис. 4. Граф IR2

На этапе оптимизации IR3 выполняются такие преобразования, как протяжка копий, удаление мертвого кода, удаление бесполезных присваиваний.

В качестве кодогенератора используется транслятор IR3 в программу на языке C#. В дальнейшем предполагается реализация кодогенератора, осуществляющего трансляцию представления IR3 в псевдокод IL платформы Microsoft .NET.

Пример. На рис. 4 показано представление IR2 для следующего фрагмента Sisal-программы (функции, вычисляющей знак числа):

```
function sgn(N: integer returns integer)
  if     N > 0 then  1
  elseif N < 0 then -1
  else               0
  end if
end function
```

Распечатка представления IR3, полученного для функции *sign* после трансляции IR2 → IR3, выглядит так:

```
  0    entry "function sgn[integer]" (V_1(I32)  returns V_3(I32));
       {
  1        V_5(I32) = V_1(I32);
  2        V_5(I32) = V_1(I32);
```

```
 3        V_9(I32) = 0x0(I32);
 4        V_13(I32) = 0x0(I32);
 5        V_7(BOOL) = (V_9(I32) < V_5(I32));
 6        V_11(BOOL) = (V_5(I32) < V_13(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
10            V_15(I32) = 0x1(I32);
11            V_3(I32) = V_15(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
15                V_19(I32) = 0x1(I32);
16                V_17(I32) =  - V_19(I32);
17                V_3(I32) = V_17(I32);
              }
              else
              {
18                V_21(I32) = 0x0(I32);
19                V_3(I32) = V_21(I32);
              }
          }
20        return;
      }
```

Содержимое IR3 после оптимизации:

```
 0    entry "function sgn[integer]" (V_1(I32)  returns V_3(I32));
      {
 5        V_7(BOOL) = (0x0(I32) < V_1(I32));
 6        V_11(BOOL) = (V_1(I32) < 0x0(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
11            V_3(I32) = 0x1(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
17                V_3(I32) =  - 0x1(I32);
              }
              else
              {
19                V_3(I32) = 0x0(I32);
              }
          }
20        return;
      }
```

ЗАКЛЮЧЕНИЕ

В статье показаны некоторые проблемы, связанные с оптимизацией программ, написанных на функциональных языках. Приведен обзор технологий, применяемых для оптимизации работы с памятью. Показаны преимущества использования внутренних представлений среднего уровня, ориентированных на оптимизацию распределения динамической памяти. Также дано краткое описание промежуточного представления среднего уровня, разработанного и реализованного для компилятора языка Sisal версии 3.1.

Важнейшим направлением дальнейшей разработки компилятора языка Sisal 3.1 является поддержка параллелизма вычислений. Для этого необходимо реализовать выделение параллельных регионов в IR2 и поддержку параллельного кода на уровне IR3.

В направлении оптимизирующих преобразований планируется реализация оптимизации ошибочных значений. Это позволит избежать проверок ошибочности значений для каждой операции и значительно повысит производительность получаемого кода.

Другим важным направлением работы видится создание полноценного кодогенератора для платформы .NET.

СПИСОК ЛИТЕРАТУРЫ

1. Allen R., Kennedy K. Optimizing Compilers for Modern Architectures. — Morgan Kaufmann, 2002.
2. Muchnik S. Compiler Design and Implementation. — Morgan Kaufmann, 1997.
3. Vijayaraghavan V., Kavi K., Shirazi B. Control Flow Extensions To The Dataflow Language SISAL. — IEEE, 1991.
4. Gharachorloo K., Sarkar V., Hennessy J. A Simple and Efficient Implementation Approach for Single Assignment Languages. — ACM, 1988.
5. Welcome M., Skedzielewski S., Yates R., Ranelletti J. IF2: An Applicative Language Intermediate Form with Explicit Memory Management. — Manual M-195. — University of California Lawrence Livermore National Laboratory, 1986.
6. Cann D. C. Compilation Techniques for High Performance Applicative Computation — PhD thesis, Colorado State University, 1989.
7. Стасенко А. П. Система интерфейсов транслятора во внутреннее представление IR1 // Методы и инструменты конструирования и оптимизации программ. — Новосибирск, 2005. — С. 229–238.
8. Стасенко А. П., Синяков А. И. Базовые средства языка Sisal 3.1. — Новосибирск, 2006. — 60 с. — (Препр. / РАН. Сиб. Отд-е. ИСИ; № 110).

А.П. Стасенко

АВТОМАТНАЯ МОДЕЛЬ ВИЗУАЛЬНОГО ОПИСАНИЯ СИНТАКСИЧЕСКОГО РАЗБОРА*

1. ВВЕДЕНИЕ

В настоящее время синтаксический разбор языков программирования [1] обычно осуществляется с помощью автоматов, сгенерированных по грамматике языков системами построения трансляторов (СПТ). При этом в СПТ, ориентированных на восходящий разбор таких, как YACC [2] или Bison [3], возникают сложности с написанием семантических правил и читабельностью сгенерированного распознавателя [4], тогда как СПТ, ориентированные на нисходящий разбор такие, как ANTRL [5], допускают часто неприемлемо узкий класс LL_k-языков [6]. Существуют подходы, комбинирующие сильные стороны восходящих и нисходящих разборов [7, 8], но все они так или иначе требуют, чтобы входная грамматика принадлежала к определённому классу, и затем генерируют распознающий автомат в виде таблицы и/или программного кода.

Требование принадлежности грамматики к определённому классу связано с необходимостью преобразования (часто ручного) исходной грамматики для получения допустимой грамматики, что, во-первых, не всегда возможно, во-вторых, приводит к отрыву от обычно более наглядного пользовательского описания языка и в-третьих, может быть источником ошибок [9]. Генерация автомата также приводит к отрыву от исходного описания языка, так как обычно неизбежно наступает этап, когда необходимо внести изменения в уже сгененированный код. Положение особенно осложняется при необходимости частого внесения изменений в исходный язык, требующих повторных ручных преобразований грамматики и перегенерации автомата.

Указанные проблемы особенно обостряются в трансляторах промышленного уровня и обычно решаются с помощью вручную написанных распознавателей. Существуют работы по ручному заданию восходящих распознавателей [10], но большее распространение получили распозна-

*Работа частично поддержана Российским фондом фундаментальных исследований (грант РФФИ № 07-07-12050).

ватели[1], основывающиеся на намного более естественном нисходящем разборе, использующем контекстную информацию там, где это необходимо для преодоления ограниченности класса LL_k-языков. Недостатками вручную написанных трансляторов обычно являются

- недостаточно сильное разделение разбора синтаксиса и семантики;
- плохое сочетание кода разбора синтаксиса, представляющего собой большие конструкции выбора, с кодом семантического разбора;
- использование больших линейно-проверяемых конструкций выбора вместо логарифмического поиска по упорядоченному множеству условий;
- сложность ручного устранения снижающих эффективность переходов по умолчанию в конструкциях выбора;
- трудность ручной минимизации числа конструкций выбора.

В данной статье предлагается автоматная модель визуального описания синтаксического разбора, которая, во-первых, позволяет избежать недостатков существующих СПТ путём наглядного описания синтаксиса языка в непосредственно исполняемом виде, и во-вторых позволяет упростить и повысить эффективность трансляторов, разрабатываемых с её помощью, путём устранения указанных выше недостатков написанных вручную трансляторов.

Предлагаемая автоматная модель наглядна, поскольку она задаётся не табличным способом, как классические магазинные автоматы, осуществляющие синтаксический анализ [6], а с помощью графа, сравнимого по сложности с графом конечного автомата, тогда как существующие способы графического задания магазинных автоматов [11, 12] требуют громоздких пометок на дугах, соответствующих переходам. Графические способы задания синтаксиса языка достаточно удобны, что подтверждается синтаксическими диаграммами Вирта [13], с помощью которых был описан язык Паскаль. Предлагаемая автоматная модель,

[1]Многие промышленные компиляторы языка Си++ и Си используют компилятор переднего плана Edison Design Group (http://www.edg.com), основанный на вручную написанном нисходящем распознавателе. Компиляторы Open Watcom также используют вручную написанные нисходящие распознаватели (http://www.openwatcom.org/index.php/Compiler_Architecture). Компилятор переднего плана G++ начиная с версии 3.4 перестал использовать СПТ YACC и был переписан вручную для реализации нисходящего разбора, для повышения эффективности трансляции и устранения конфликтов распознавателя (http://gcc.gnu.org/gcc-3.4/changes.html).

по сути, является развитием и уточнением идеи, заложенной в синтаксических диаграммах Вирта.

Статья состоит из десяти разделов. В разд. 2 вводится определение модели γ-автомата и его частных случаев: γ_s-автоматов, γ_1-автоматов и γ_{s1}-автоматов, графическое изображение которых описывается далее в разделе 3. Наибольший интерес представляет класс γ_1-автоматов, допускающих детерминированное исполнение и класс контекстно-зависимых языков, однако для полноты картины в разд. 4 исследуется класс языков, представимых γ_s-автоматами. В двух следующих разделах рассмотрение сосредотачивается на γ_{s1}-автоматах, как важного частного случая γ_1-автоматов. В разд. 5 доказывается существование класса грамматик, эквивалентного классу γ_{s1}-автоматов. В разд. 6 исследуется класс языков, представимых γ_{s1}-автоматами, и показывается его ограниченность. В разд. 7 рассматривается соответствие между классами языков, классом γ_1-автоматов и некоторых его подклассов. В разд. 8 рассматриваются способы оптимизации γ_1-автоматов. Статья завершается разделом 9, в котором описываются преимущества реализации транслятора, синтаксический разбор которого описывается с помощью графического представления γ_1-автоматов.

2. ОПРЕДЕЛЕНИЕ МОДЕЛИ

Вводимая далее модель автомата для визуального описания синтаксического разбора (далее просто γ-автомата) основывается на классической модели магазинного автомата [14], на которую наложены ограничения на вид функции перехода, призванные повысить наглядность её графического представления. С другой стороны, модель γ-автомата содержит дополнения, направленные на расширение класса языков, представимых автоматами модели и дальнейшее повышение читабельности её графического представления.

Модель γ-автомата определяется кортежем $A = (T,\ S,\ S_k,\ s_0,\ S_{end},\ K,\ k_0,\ \Phi,\ \mu,\ \tau,\ \tau_k,\ T_k,\ f_e)$, где T — конечный входной алфавит, S — конечный алфавит состояний, $S_k \subseteq S$ — множество состояний с контекстным переходами, $s_0 \in S$ – начальное состояние, $S_{end} \subseteq S$ — множество конечных состояний, K — множество контекстов, $k_0 \in K$ — начальный контекст, Φ – множество контекстных функций вида $K \to K \times T_k$, $\mu : S_k \to \Phi$ — функция разметки состояний контекстными функци-

ями, $\tau : (S \setminus S_k) \times (T \cup \{\epsilon^2\}) \times M \to 2^{S \times M^*}{}^3$ – функция переходов, $\tau_k : S_k \times T_k \to S$ — функция контекстных переходов, T_k – множество пометок контекстных переходов, $f_e : (S \setminus S_k) \times M_e \to M_e^*$ — функция изменения магазина исключений M_e[4].

Функция переходов γ-автомата τ для каждого состояния $s_1 \in S \setminus S_k$ и символа перехода $\delta \in T \cup \{\epsilon\}$ имеет один из следующих трёх видов (упоминаемые далее как переходы первого, второго и третьего вида, соответственно):

1) семейство переходов $\tau(s_1, \delta, m) = (s_2, m)$ для всех $m \in M$, не зависящих от состояния и не меняющих содержимое магазина M;
2) семейство переходов $\tau(s_1, \delta, m) = (s_n, ms_2 \dots s_{n-1})$ для всех $m \in M$, не зависящих от содержимого магазина M, но добавляющих в него цепочку $s_2 \dots s_{n-1}$ (s_2 в первую очередь);
3) семейство переходов $\tau(s_1, \delta, s_2) = (s_2, \epsilon)$ для всех $s_2 \in M$ в состояние, вытолкнутое из магазина M.

Функция f_e для каждого состояния $s \in S \setminus S_k$ имеет один из следующих четырёх видов:

- функция вида $f_e(s, m_e) = (m_e)$ для всех $m_e \in M_e$, не зависящая от состояния и не меняющая содержимое магазина M_e;
- функция вида $f_e(s, m_e) = (m_e s_e)$ для всех $m_e \in M_e$, добавляющая состояние s_e в магазин M_e;
- функция вида $f_e(s, m_e) = (s_e)$ для всех $m_e \in M_e$, заменяющая состояние m_e на s_e на вершине магазина M_e;
- функция вида $f_e(s, m_e) = (\epsilon)$ для всех $m_e \in M_e$, выталкивающая состояние из магазина M_e.

Автомат читает символы множества входной ленты с алфавитом T, в котором выделен специальный символ t_0, находящийся в конце ленты. Автомат имеет магазин M и магазин исключений M_e, содержащие символы из множества состояний $S \setminus S_k$. Конфигурацией автомата называется элемент множества $T^* \times S \times M^* \times M_e^* \times K$. Работа автомата определяется сменой конфигураций от начальной конфигурации $(\omega_1, s_0, \epsilon, \epsilon, k_0)$, где ω_1 обозначает начальную цепочку символов, до ко-

[2]Символ ϵ обозначает пустую цепочку.

[3]Звёздочка после множества X обозначает множество цепочек в алфавите X, включающее пустую цепочку.

[4]Помимо основного магазина M, автомат содержит дополнительный магазин исключений M_e.

нечной конфигурации, принадлежащей множеству $T^* \times S_{end} \times M^* \times M_e^* \times K$.

Для состояния $s \in S \setminus S_k$ смена конфигурации (ω, s, m, m_e, k) на конфигурацию $(\omega', s', m', m'_e, k')$ определяется следующим образом: $m'_e = f_e(s, m_e)$, $k' = k$, $(s', m') = \tau(s, \delta = \{t, \epsilon\}, m)$, где если $\delta = t$, то $\omega = t\omega'$, иначе $\omega' = \omega$. Если функция переходов τ неопределена для текущей конфигурации, то новая конфигурация определяется следующим образом: $\omega' = \omega$, состояние s' равно состоянию на вершине магазина $m''_e = f_e(s, m_e)$[5], $m' = m$, m'_e равно содержимому магазина m''_e с вытолкнутым состоянием.

Для состояния $s \in S_k$ смена конфигурации (ω, s, m, m_e, k) на конфигурацию $(\omega', s', m', m'_e, k')$ определяется иначе. Новое состояние s' определяется функцией переходов τ_k как $s' = \tau_k(s, t_k)$, где t_k и новое состояние контекста k' определяются функцией ϕ на основании текущего контекста $(t_k, k') = \phi(k)$, где функция $\phi \in \Phi$, определяется функцией разметки μ для текущего состояния $\phi = \mu(s)$. Содержимое магазинов и входной ленты не изменяется: $m' = m$, $m'_e = m_e$ и $\omega' = \omega$.

Определение 1. *Языком γ-автомата называется множество цепочек, начиная работу с которых, автомат достигает конечного состояния.*

Введём подклассы класса γ_s-автоматов.

Определение 2. *Под γ_s-автоматом подразумевается γ-автомат, в котором $S_k = \emptyset$ и все функции f_e не зависят и не меняют содержимое магазина M_e.*

Заметим, что у γ_s-автомата можно не задавать компоненты K, k_0, Φ, μ, τ_k и T_k, а в конфигурации автомата указывать только первые три элемента: (T^*, S, M^*). Класс γ_s-автоматов интересен тем, что он является подклассом обычных недетерминированных магазинных автоматов с более ограниченным видом функции переходов τ, что, однако, как показано в разделе 4, не повлекло за собой сужения класса представимых ими языков.

Определение 3. *В γ-автомате переходом функции переходов τ называется множество $\tau(s, \delta, m)$.*

[5]Если $m_e = \epsilon$, то поведение γ-автомата не определено.

Определение 4. *В γ-автомате переход $\tau(s, \delta, m)$ функции переходов τ называется детерминированным, если $|\tau(s, \delta, m)| \leq 1$.*

Определение 5. *Под γ_1-автоматом (детерминированным γ-автоматом) подразумевается γ-автомат, в котором все переходы функции переходов τ детерминированы, а неоднозначность выбора между переходами $\tau(s_1, \epsilon, m)$ и $\tau(s_1, t, m)$ разрешается предпочтением перехода по t.*

Класс γ_1-автоматов важен тем, что такие автоматы допускают детерминированное исполнение и позволяют разбирать широкий класс контекстно-зависимых языков, что доказывается в разделе 7.

Определение 6. *Пересечение класса γ_s-автоматов и класса γ_1-автоматов обозначается как класс γ_{s1}-автоматов.*

Класс γ_{s1}-автоматов интересен тем, что он является подклассом обычных детерминированных магазинных автоматов с более ограниченным видом функции переходов τ, что, как показано в разделе 6, повлекло за собой сужение класса представимых ими языков.

Рассмотрим примеры γ-автоматов, первый из которых задаёт язык контекстно-свободной грамматики. Под грамматикой G подразумевается четвёрка $\{T, N, s_0, R\}$, где T — это терминальный алфавит, N — нетерминальный алфавит, $s_0 \in N$ — начальный символ грамматики, R — множество правил вывода, для контекстно-свободных грамматик имеющих вид $B \to \alpha$, где $B \in N$, $\alpha \in (N \cup T)^*$ [14]. Под языком грамматики подразумевается множество цепочек, выводимых из начального символа грамматики.

Рассмотрим примеры γ-автоматов. Начнём с примера γ_{s1}-автомата, задающего язык контекстно-свободной грамматики. Под грамматикой G подразумевается четвёрка $\{T, N, s_0, R\}$, где T — это терминальный алфавит, N — нетерминальный алфавит, $s_0 \in N$ — начальный символ грамматики, R — множество правил вывода, для контекстно-свободных грамматик имеющих вид $B \to \alpha$, где $B \in N$, $\alpha \in (N \cup T)^*$ [14]. Под языком грамматики подразумевается множество цепочек, выводимых из начального символа грамматики.

В качестве контекстно-свободного языка возьмём скобочной язык L_{s1}, задаваемый грамматикой $G_{s1} = (\{\text{‘(’}, \text{‘)’}\}, \{S, A\}, S, R)$, где $R = \{\text{“}S \to (A)\text{”}, \text{“}A \to S\text{”}, \text{“}A \to SA\text{”}\}$. Компоненты γ_{s1}-автомата A_{s1}, задающего язык L_{s1}, определяются так: $T = \{\text{‘(’}, \text{‘)’}, t_0\}$, $S = \{s_0, s_1, s_2, s_{end}\}$, $S_{end} = \{s_{end}\}$. Функция переходов τ определяется следующим образом:

$\tau(s_0, \text{‘(’}, m) = (s_1, ms_{end})$, $\tau(s_1, \text{‘(’}, m) = (s_1, ms_2)$, $\tau(s_1, \text{‘)’}, s) = (s, \epsilon)$, $\tau(s_2, \epsilon, m) = (s_1, m)$[6].

Если на входной ленте находится допустимая цепочка ‘(()())’, то работа распознающего её γ_{s1}-автомата A_{s1} будет определяться следующей последовательностью конфигураций (конфигурации упорядочены слева направо, сверху вниз):

‘(()())’,	s_0, ϵ),	‘()())’,	s_1, s_{end}),	‘))’,	$s_1, s_{end}\ s_2$),
‘())’,	s_2, s_{end}),	‘())’,	s_1, s_{end}),	‘))’,	$s_1, s_{end}\ s_2$),
‘)’,	s_2, s_{end}),	‘)’,	s_1, s_{end}),	$(\epsilon$,	s_{end}, ϵ).

Рассмотрим пример γ_1-автомата, разбирающего контекстно-зависимый язык L_1. Зададим язык L_1, изменив скобочный язык L_{s1} дополнительным требованием того, что содержимое скобок с номером n в скобках с уровнем вложенности равным n может содержать цепочки произвольного алфавита не содержащего скобок. В то же время данные цепочки предполагаются принадлежащими некоторому контекстно-зависимому языку L_e, который необходимо попытаться разобрать и, в случае неудачи, пропустить оставшуюся часть цепочки. Данная задача очень близка к реальной задаче разбора языка программирования с восстановлением разбора после синтаксических ошибок.

Пусть γ_1-автомат A_e, распознающий язык L_e, имеет следующие компоненты: (T', S', S'_k, s'_0, S'_{end}, K', k'_0, Φ', μ', τ', τ'_k, T'_k, f'_e). Компоненты γ_1-автомата A_1, распознающего язык L_1, определяются так (в предположении, что все множества, участвующие в последующих объединениях, не пересекаются): $T = \{\text{‘(’}, \text{‘)’}, t_0\} \cup T'$, $S = \{s_0, \ldots, s_7, s_{end}\} \cup S'$, $S_k = \{s_2\} \cup S'_k$, $S_{end} = \{s_{end}\}$, $K = \{\text{содержимое магазина чисел } L\} \times K'$, $k_0 = (\text{магазин чисел } L, \text{содержащий число } 1, k'_0)$, $\Phi = \{\phi\} \cup \Phi'$, $T_k = \{true, false\} \cup T'_k$.

Функция μ автомата A_1 для состояния s_2 определяется как $\mu(s_2) = \phi$, а для состояний из множества S'_k — тождественна функции μ'. Функция переходов τ_k для состояния s_2 определяется как $\tau_k(s_2, true) = s_3$, $\tau_k(s_2, false) = s_5$, а для состояний из множества S'_k – тождественна функции τ'_k. Функция f_e для состояний из множества $S \setminus S_k$ определяется как $f_e(s_3, m_e) = (m_e s_7)$, $f_e(s_4, m_e) = (\epsilon)$, а для состояний из множества $S' \setminus S'_k$ — тождественна функции f'_e.

Функция ϕ автомата A_1 возвращает значение $true$, если число на

[6]Правила $\tau(s_1, \text{‘(’}, m) = (s_1, ms_2)$ и $\tau(s_2, \epsilon, m) = (s_1, m)$ можно было бы заменить правилом $\tau(s_1, \text{‘(’}, m) = (s_1, ms_1)$, но это не было сделано для поддержания сходства с рассматриваемыми далее γ-схемами, которые не могут непосредственно представить указанное правило.

вершине магазина L равно глубине магазина L, и возвращает значение $false$ иначе. Также (после проверки предыдущего равенства) функция ϕ увеличивает на единицу число на вершине магазина L. Для целей реальной трансляции состояниям и переходам модели γ-автомата приписываются пометки, обозначающие различные транслирующие процедуры над контекстом K, что более подробно рассматривается в разделе 9. Тем самым вводятся функция ϕ_{push}, которая добавляет к магазину L единицу, и функция ϕ_{pop}, которая выталкивает число из магазина L. Состояние s_5 помечено функцией ϕ_{push}.

Функция переходов τ автомата A_1 для состояний из множества $S' \backslash S'_k$ тождественна функции перехода τ', а для состояний из $S \setminus S_k$ определяется следующим образом: $\tau(s_0, \text{`('}, m) = (s_1, ms_{end})$, $\tau(s_1, \text{`('}, m) = (s_2, m)$, $\tau(s_1, \text{`)'}, s) = (s, \epsilon)$ (переход помечен функцией $\phi_{pop}$), $\tau(s_3, \epsilon, m) = (s'_0, m)$, $\tau(s_4, \epsilon, m) = (s_1, m)$, $\tau(s_5, \epsilon, m) = (s_1, ms_6)$, $\tau(s_6, \epsilon, m) = (s_1, m)$, $\tau(s_7, t, m) = (s_7, m)$ для всех $t \in T \setminus \{\text{`)'}\}$, $\tau(s_7, \text{`)'}, m) = (s_1, m)$, $\tau(s'_{end}, \text{`)'}, m) = (s_4, m)$ для всех $s'_{end} \in S'_{end}$.

Если цепочка '*abc*' не принадлежит языку L_e, и на входной ленте находится допустимая цепочка '((*abc*))', то работа распознающего её γ_1-автомата A_1 будет определяться следующей последовательностью конфигураций:

('((*abc*))',	$s_0, \epsilon, \epsilon, 1$[7]),	('(*abc*))',	$s_1, s_{end}, \epsilon, 1$),	('*abc*))',	$s_2, s_{end}, \epsilon, 1$),
('*abc*))',	$s_3, s_{end}, \epsilon, 2$),	('*abc*))',	$s'_0, s_{end}, s_7, 2$),	... [8],	
('ω^9))',	$s_7, s_{end}, \epsilon, 2$),	... [10],		('))',	$s_1, s_{end}, \epsilon, 2$),
('))',	$s_2, s_{end}, \epsilon, 2$),	('))',	$s_5, s_{end}, \epsilon, 3$),	('))',	$s_1, s_{end}\ s_6, \epsilon, 3\ 1$),
(')',	$s_6, s_{end}, \epsilon, 3$),	(')',	$s_1, s_{end}, \epsilon, 3$),	(ϵ,	$s_{end}, \epsilon, \epsilon, \epsilon$).

3. ИЗОБРАЖЕНИЕ МОДЕЛИ

Для визуального представления модели γ-автомата было разработано графическое представление [15], называемое γ-схемой, которое отображает модель γ-автомата в виде ориентированного размеченного графа. Ниже перечислены классы вершин графа γ-схемы.

[7] Единицей обозначается содержимое магазина L.

[8] Многоточием обозначается работа автомата A_e, который не сможет разобрать цепочку '*abc*', то есть попадёт в неопределённое состояние.

[9] Символом ω обозначается цепочка из языка $\{\epsilon, \text{`}c\text{'}, \text{`}bc\text{'}, \text{`}abc\text{'}\}$.

[10] Многоточием обозначается повторение предыдущей конфигурации с уменьшающейся по одному символу входной строкой до тех пор, пока не исчерпается цепочка ω.

1. Прямоугольные вершины, соответствующие состояниям $s \in S$:
   (a) вершины, соответствующие состояниям $s \in S \setminus S_k$:
      i. вершины со сплошной границей, для состояний которых функция f_e имеет вид $f_e(s, m_e) = (m_e)$ и $f_e(s, m_e) = (m_e s_e)$ (рис. 1(а));
      ii. вершины со штриховой границей, для состояний которых функция f_e имеет вид $f_e(s, m_e) = (s_e)$ и $f_e(s, m_e) = (\epsilon)$ (рис. 1(б));
   (b) вершины, соответствующие состояниям из S_k и имеющие сплошную границу и дополнительную пометку элементом множества $\phi \in \Phi$ (рис. 1(в)).
2. Круглые вершины v с пометкой "pop", задающие переходы третьего типа $\tau(s, \delta, s_1) = (s_1, \epsilon)$ из состояний s, вершины которых имеют исходящую дугу, которая входит в вершину v и имеет различный вид в зависимости от символа δ (рис. 1(м,н,о)).

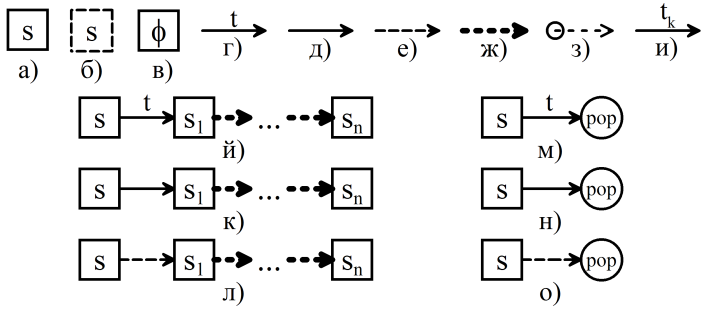


Рис. 1. Базовые элементы γ-схемы

Дуги γ-схем также подразделяются на несколько классов.

1. Дуги из вершин, соответствующих состояниям $s \in S \setminus S_k$:
   (a) дуги, задающие переходы первого типа различным способом в зависимости от второго аргумента δ функции переходов τ:
      i. сплошная дуга с меткой $t \in T$ для $\delta \in T$ (рис. 1(г));
      ii. сплошная дуга без метки для $\delta \in T \setminus T_s$, где где T_s — это метки всех других помеченных дуг, исходящих из вершины состояния s (рис. 1(д))
      iii. штриховая дуга без метки для $\delta = \epsilon$ (рис. 1(е));
   (b) непомеченные пунктирные дуги без символа круга в начале

дуги, образующие путь от вершины состояния s_1 до вершины состояния s_n и задающие переходы второго типа $\tau(s, \delta, m) = (s_n, ms_1 \dots s_{n-1})$ из состояния s, из вершины которого исходит дуга, которая входит в вершину состояния s_1 и имеет различный вид в зависимости от символа δ (рис. 1(й,к,л));

(c) непомеченные пунктирные дуги с символом круга в начале дуги, ведущие из вершины состояния s в вершину состояния s_e, которые участвуют в функции f_e вида $f_e(s, m_e) = (m_e s_e)$ и $f_e(s, m_e) = (s_e)$ (рис. 1(з)).

2. Сплошные дуги с меткой $t_k \in T_k$, исходящие из вершин, соответствующих состояниям из S_k (рис. 1(и)).

Если γ-схема задаёт γ_1-автомат, то она называется γ_1-схемой, и на неё накладывается несколько дополнительных условий. Ни из одной вершины не может исходить несколько одинаково помеченных дуг. Из вершины, соответствующей состоянию из $S \setminus S_k$, не может исходить более одной непомеченной сплошной или штриховой дуги, более одной пунктирной дуги без символа круга в начале дуги и более одной пунктирной дуги с символом круга в начале дуги.

Рассмотрим примеры γ-схем. На рис. 2 приведена γ_{s1}-схема, задающая γ_{s1}-автомат A_{s1} из раздела 2. На рис. 3 приведена γ_1-схема, задающая γ_1-автомат A_1 из разд. 2. Нетрудно заметить, что рис. 3 более компактно и наглядно задаёт описание автомата A_1, занимающее половину страницы в текстовом виде.

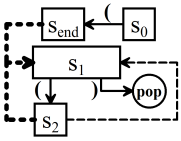


Рис. 2. Пример γ_{s1}-схемы

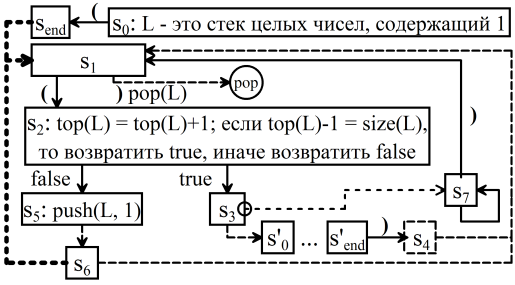


Рис. 3. Пример γ_1-схемы

4. СВОЙСТВА γ_S-АВТОМАТА

В данном разделе показывается, что γ_s-автоматы по мощности представимых языков равны классу контекстно-свободных языков. Таким образом, введённые ограничения на функцию переходов τ не повлекли за собой сужения класса языков, представимых недетерминированными магазинными автоматами.

Теорема 1. *Любой контекстно-свободный язык [14] можно задать γ_s-автоматом.*

Доказательство. Возьмём контекстно-свободную грамматику $G = \{T', N', s'_0, R\}$. Построим γ_s-автомат A, такой что $L_A = L_G$. Определим компоненты γ_s-автомата A следующим образом. Входной алфавит определим как $T = T' \cup \{t_0\}$. Множество состояний $S = N' \cup N'' \cup \{s_{end}\} \cup \{s_0\}$, где $N'' = \{r_{i,j}\}$, $i = 1 \dots |R|$, $j = 1 \dots J$, где J — это длина правой части i-го правила вывода, $N' \cap N'' = \emptyset$, $s_{end} \notin N' \cup N''$, $s_0 \notin N' \cup N''$, $s_{end} \neq s_0$. В качестве конечного состояния возьмём $S_{end} = \{s_{end}\}$. Функция переходов τ содержит переход из начального состояния $\tau(s_0, \epsilon, \epsilon) = (s'_0, s_{end})$, добавляющий конечное состояние s_{end} в магазин M.

В качестве начального состояния входной ленты возьмём произвольную цепочку языка L_G. Для каждого i-го правила грамматики $B_i \to \alpha_{i,1} \dots \alpha_{i,J}$, где $\alpha_{i,1\dots J} \in N' \cup T'$, добавим переходы к функции переходов τ.

1. Для каждого $j = 1 \dots J$ полагаем следующее:
   (a) если $\alpha_{i,j} \in T'$, то добавляем переход

$$\tau(r_{i,j-1}, \alpha_{i,j}, m) = (r_{i,j}, m),$$

   полагая, что $r_{i,0} = B_i$;
   (b) если $\alpha_{i,j} \in N'$, то добавляем переход

$$\tau(r_{i,j-1}, \epsilon, m) = (\alpha_{i,j}, m r_{i,j}).$$

2. Добавляем переход в состояние на вершине магазина

$$M : \tau(r_{i,J}, \epsilon, s) = (s, \epsilon).$$

Тем самым, одно правило магазинного автомата, разбирающего данную грамматику G, заменено несколькими правилами γ_s-автомата A с помощью дополнительных магазинных символов. ■

Множества языков, задаваемых γ_s-автоматами и контекстно-свободными грамматиками, равны, так как любой γ_s-автомат задаёт контекстно-свободный язык, ввиду очевидного факта, что любой γ_s-автомат является магазинным автоматом. К сожалению, ввиду недетерминированности γ_s-автоматы не допускают эффективное[11] исполнение и поэтому не интересны для применений в реальных трансляторах [14].

5. СВЯЗЬ γ_{S1}-АВТОМАТА С КЛАССАМИ ГРАММАТИК

В данном разделе показывается, что существует класс грамматик, эквивалентный классу γ_{s1}-автоматов и не совпадающий с известным классом LL_1-грамматик. По определению, классы автоматов или грамматик эквивалентны между собой, если для каждого объекта A одного класса, найдётся объект в другом классе, задающий язык объекта A.

Определим класс грамматик $G_\gamma = (T, N, s_0, R)$ с множеством правил R вида: (1) "$N_1 \to t_1\alpha$", (2) "$N_1 \to N_2 t_1 \alpha$" и (3) "$s_0 \to \epsilon$", где $N_{1,2} \in N$, $t_1 \in T$ и $\alpha \in (N \cup T)^*$, со следующими ограничениями:

1) все правила множества R с одинаковым нетерминалом N_1 имеют один вид;
2) каждое правило множества R с одинаковым нетерминалом N_1 уникально по терминалу t_1;
3) каждое правило второго вида множества R с одинаковым нетерминалом N_1 имеет одинаковый нетерминал N_2.

Покажем, что класс грамматик G_γ эквивалентен классу γ_{s1}-автоматов.

Теорема 2. *Любой грамматике из класса G_γ соответствует γ_{s1}-автомат, задающий язык этой грамматики.*

Доказательство. Рассмотрим доказательство теоремы 1 для грамматики из G_γ в месте построения функции переходов γ_s-автомата A. Докажем, что γ_s-автомат A является γ_1-автоматом, так как все переходы функции переходов τ автомата A для грамматики из G_γ детерминированы. Недетерминированность переходов может возникнуть в пунктах 1.a и 1.b теоремы 1 при $j = 1$, когда $r_{i,0} = B_i$. В пункте 1.a переход $\tau(r_{i,0}, \alpha_{i,1}, m) = (r_{i,1}, m)$ будет детерминированным ввиду того, что в грамматике из G_γ каждое правило с одинаковым нетерминалом $r_{i,0}$ в левой части имеет уникальный терминал $\alpha_{i,1}$. В пункте 1.b правило перехода $\tau(r_{i,0}, \epsilon, m) = (\alpha_{i,1}, m r_{i,1})$ будет детерминировано, так как

[11] Исполнение за время, пропорциональное длине входной цепочки.

- в грамматике из G_γ все правила с одинаковым нетерминалом $r_{i,0}$ в левой части начинаются с одинакового нетерминала $\alpha_{i,1}$ в правой части, что напрямую следует из третьего ограничения в определении класса грамматик G_γ;
- состояния $r_{i,1}$ для совпадающих $r_{i,0}$ нетерминалов можно объединить в одно, так как переход из них однозначно определяется терминалом $\alpha_{i,2}$, что напрямую следует из второго ограничения в определении класса грамматик G_γ.

Очевидно, что автомат A является γ_{s1}-автоматом, так как оставшийся переход в родительское состояние пункта 2 детерминирован, ввиду уникальности состояния $r_{i,J}$. ■

Теорема 3. *Любому γ_{s1}-автомату соответствует грамматика из G_γ, задающая язык этого автомата.*

Доказательство. Возьмём γ_{s1}-автомат $A = \{T, S, s_0, S_{end}, \tau\}$. Согласно теореме 11, устраним в автомате A мнимые переходы первого и второго видов. Построим грамматику $G_\gamma = (T, S, s_0, R)$. Для каждого перехода первого вида $\tau(s_1, t, m) = (s_2, m)$ к множеству R добавим правило "$s_1 \to ts_2$". Для каждого перехода второго вида $\tau(s_1, t, m) = (s_n, ms_2 \dots s_{n-1})$ к множеству R добавим правило "$s_1 \to ts_n s_{n-1} \dots s_2$". Для каждого перехода третьего вида $\tau(s_1, t, s_2) = (s_2, \epsilon)$ к множеству R добавим правило "$s_1 \to t$". Для каждого перехода третьего вида $\tau(s_1, \epsilon, s_2) = (s_2, \epsilon)$ к множеству R добавим правило "$s_1 \to \epsilon$". Удалим мнимые переходы из грамматики, используя известный алгоритм для контекстно-свободных грамматик [14]. Очевидно, что каждое правило множества R с одинаковым нетерминалом s_1 в левой части имеет уникальный терминал t, по свойству γ_{s1}-автомата.

■

Покажем, что класс грамматик G_γ не совпадает с классом LL_1-грамматик.

Определение 7. *Грамматика принадлежит классу LL_1 [6], если множества ВЫБОР для правил грамматики с одинаковой левой частью не пересекаются.*

Определение 8. *Множество ВЫБОР правила грамматики по определению содержит терминальные символы, при появлении которых под читающей головкой распознаватель должен применять это правило.*

Теорема 4. *Существуют LL_1-грамматики, не принадлежащие классу G_γ.*

Доказательство. Рассмотрим грамматику $G = (T = \{a, b, c\}, N = \{S, A, B\}, S, R)$, где $R = \{$“$S \to Ac$”, “$S \to Bc$”, “$A \to a$”, “$B \to b$”$\}$. Грамматика G, очевидно, принадлежит классу LL_1, но не принадлежит классу G_γ, так как в ней существуют правила с одинаковой левой частью, но различными нетерминалами в начале правой части. ■

Теорема 5. *В классе G_γ существуют грамматики, не принадлежащие классу LL_1.*

Доказательство. Рассмотрим грамматику $G = (T = \{a, b, c, d\}, N = \{S, A\}, S, R)$, где $R = \{$“$S \to Ac$”, “$S \to Ad$”, “$A \to a$”, “$A \to b$”$\}$. Грамматика G удовлетворяет всем требованиям класса грамматик G_γ, но не принадлежит классу LL_1, так как содержит неоднозначность по первому терминалу выбора правила с нетерминалом S в левой части. ■

6. СВЯЗЬ γ_{S1}-АВТОМАТА С КЛАССАМИ ЯЗЫКОВ

В этом разделе показывается, что множества языков, задаваемых γ_{s1}-автоматами и LL_1-грамматиками, равны. Тем самым, введённые ограничения на функцию переходов τ повлекли за собой сужение класса детерминированных контекстно-свободных языков (представимых детерминированными магазинными автоматами) до класса LL_1-языков.

Определение 9. *Язык принадлежит классу LL_1, если существует LL_1-грамматика, его задающая.*

Теорема 6. *Любой LL_1-язык можно задать γ_{s1}-автоматом.*

Доказательство. Рассмотрим LL_1-грамматику G произвольного LL_1-языка. Изменим грамматику G так, чтобы она продолжала задавать язык исходной грамматики, но принадлежала классу G_γ. Этого достаточно, так как уже доказано существование γ_{s1}-автомата, задающего язык L_{G_γ}. В грамматике G множества ВЫБОР, построенные для правил с одинаковой левой частью, не содержат одинаковых элементов, поэтому очевидно, что каждое правило грамматики G можно заменить множеством правил первого вида $N_1 \to t_1\alpha$ для $t_1 \in$ ВЫБОР правил грамматики класса G_γ, путём замены первого нетерминала правой части правила. Очевидно, что после таких замен грамматика останется в классе LL_1 и непересечение множеств ВЫБОР для правил с одинаковой левой частью будет означать соблюдение второго ограничения

в определении класса грамматик G_γ. Первое и второе ограничения в определении класса грамматик G_γ для изменённой грамматики будут очевидно удовлетворены. ■

Теорема 7. *Любой γ_{s1}-автомат задаёт LL_1-язык.*

Доказательство. Рассмотрим грамматику из G_γ, соответствующую взятому γ_{s1}-автомату. Правила первого вида грамматики из G_γ с одинаковой левой частью имеют непересекающиеся множества ВЫБОР ввиду того, что они различны по первому терминалу.

Рассмотрим правила второго вида грамматики из G_γ с одинаковой левой частью "$N_1 \to N_2 t_1 \alpha_1$". Введём новый нетерминал $N_1^{'}$ для каждого нетерминала N_1, стоящего в левой части правила второго вида. К грамматике добавим правила первого вида "$N_1^{'} \to t_1 \alpha_1$".

Если правила первого нетерминала N_2 имеют второй вид "$N_2 \to N_3 t_2 \alpha_2$", то после подстановки снова получаются правила второго вида "$N_1 \to N_3 t_2 \alpha_2 t_1 \alpha_1$". Делая ещё одну подстановку, получим правила вида "$N_1 \to N_3 t_2 \alpha_2 N_1^{'}$", которые будут удовлетворять второму ограничению из определения класса грамматик G_γ, так как для каждого нетерминала N_1 они различны по терминалу t_2, ввиду уникальности терминала t_2 для одинаковых нетерминалов N_2. Тем самым, для полученных правил можно повторять процедуру замены первого нетерминала на правила второго вида до тех пор, пока первый нетерминал не будет стоять в левой части правил первого вида.

Если правила первого нетерминала N_2 имеют первый вид "$N_2 \to t_2 \alpha_2$", то после подстановки получатся правила первого вида "$N_1 \to t_2 \alpha_2 N_1^{'}$", которые также будут удовлетворять второму ограничению из определения класса грамматик G_γ, так как для каждого нетерминала N_1 они различны по первому терминалу. Правила второго вида грамматики из G_γ с одинаковой левой частью также имеют непересекающиеся множества ВЫБОР.

Все правила грамматики с одинаковой левой частью имеют непересекающиеся множества ВЫБОР, поэтому грамматика из G_γ принадлежит классу LL_1. ■

7. СВОЙСТВА γ_1-АВТОМАТА

Как показано ранее, γ_{s1}-автоматы ограничены довольно узким классом LL_1-языков и поэтому малоприменимы для синтаксического анализа реальных языков программирования, спроектированных без учёта

принадлежности к этому классу языков. Поэтому с точки зрения мощности множества поддерживаемых языков, интерес представляет более широкий класс γ_1-автоматов.

Теорема 8. *Любой контекстно-зависимый язык можно задать γ_1-автоматом.*

Доказательство.[12] Возьмём грамматику $G = (T', N', s'_0, R')$ произвольного контекстно-зависимого языка. Известно, что существует линейно-ограниченный автомат A_G, задающий язык L грамматики G [14].

Рассмотрим γ_1-автомат, в котором $T = T' \cup \{t_0\}$, $S = \{s_0, s_1, s_2, s_3\}$, $S_{end} = \{s_3\}$, $S_k = \{s_2\}$, $K = \{\text{строка символов } L\}$, $T_k = \{true, false\}$, $\mu(s_2) = \phi_{decide}$, $\Phi = \{\phi_{decide}\}$. Состояние s_0 помечено действием "сделать строку символов L пустой". Состояние s_1 помечено действием "добавить последний прочитанный символ входной ленты к строке символов L". Функция переходов τ, определяется так: $\tau(s_0, t, m) = (s_1, m)$, $\tau(s_0, t_0, m) = (s_2, m)$, $\tau(s_1, t, m) = (s_2, m)$, $\tau(s_1, t_0, m) = (s_2, m)$, для для любого $t \in T$ и любого $m \in M$.

Функция ϕ_{decide} реализует автомат A_G, используя в качестве его ленты строку символов L и возвращая $true$, если строка допустима, и $false$ иначе. Функция переходов τ_k содержит переход $\tau_k(s_2, true) = (s_3)$, помеченный действием "сообщить, что входная цепочка принадлежит языку L". Функция переходов τ_k также содержит переход $\tau_k(s_2, false) = (s_3)$, помеченный действием "сообщить, что входная цепочка не принадлежит языку L".

Построенный автомат с очевидностью задаёт язык L, так как, по сути, он только накапливает символы входной ленты для использования её в качестве ленты линейно-ограниченного автомата. ■

Используемый в доказательстве теоремы 8 автомат не нагляден, так как скрывает все детали разбора в контекстной функции ϕ_{decide}. Однако для большинства языков программирования, синтаксис которых в основном представим детерминированным контекстно-свободным языком, можно сохранить разумный баланс наглядности автомата и его контекстных функций, как, например, на рис. 3.

Механизм иерархической обработки неопределённостей γ-автоматов, задаваемый функцией изменения магазина исключений f_e, был введён для удобства построения и сохранения наглядности реальных γ-схем,

[12] Доказательство ограничивается рассмотрением контекстно-зависимых языков в предположении, что контекст исполнения ограничен конечной памятью реальных вычислительных систем.

описывающих синтаксический анализ. Данный механизм позволяет избежать необходимости полного определения функции переходов τ, особенно в случае, когда реакция по умолчанию, которая может рассматриваться как ошибка разбора, легко унифицируема. По своей сути механизм иерархической обработки неопределённостей сходен с механизмом обработки исключений известных языков программирования.

Определение 10. *Под γ_{se1}-автоматом подразумевается γ_1-автомат, в котором $S_k = \emptyset$.*

Хотя механизм иерархической обработки неопределённостей не предназначается для расширения класса языков, представимых γ_{s1}-автоматами, можно показать, что язык γ_{se1}-автомата включает не контекстно-свободный язык и не включает контекстно-свободный язык. Таким образом, язык γ_{se1}-автомата не совпадает ни с одним из известных классов языков но, очевидно, включает в себя класс LL_1-языков.

Теорема 9. *Существует не контекстно-свободный язык, распознаваемый γ_{se1}-автоматом.*

Доказательство. Рассмотрим известный не контекстно-свободный язык, состоящий из цепочек вида $a^n b^n c^n$ для всех $n \geq 0$. Рассмотрим γ_{se1}-автомат, в котором $T = \{a, b, c, t_0\}$, $S = \{s_0, s_1, s_2, s_3, s_{yes}, s_{no}\}$, $S_{end} = \{s_{yes}\}$. Функция переходов τ определяется следующим образом: $\tau(s_0, \epsilon, m) = (s_1, ms_3)$, $\tau(s_1, a, m) = (s_1, ms_2)$, $\tau(s_1, \epsilon, m) = (m, \epsilon)$, $\tau(s_2, b, m) = (m, \epsilon)$, $\tau(s_3, c, m) = (s_3, m)$, $\tau(s_{no}, t, m, m_e) = (s_{no}, m, m_e)$ для всех $t \in T$. Функция f_e определяется так: $f_e(s_0, m_e) = (m_e s_{yes})$, $f_e(s_1, m_e) = (m_e s_{no})$, $f_e(s_3, m_e) = (\epsilon)$, $f_e(s_{no}, m_e) = (\epsilon)$.

При распознавании букв a, по сути, их количество n запоминается в магазинах M и M_e. Магазин M используется для распознавания ровно n букв b, а магазин M_e — для распознавания ровно n букв c. Автомат может достигнуть конечного состояния s_{yes} только в случае возникновения неопределённой ситуации в состоянии s_3, когда сработает переход в состояние на вершине магазина M_e, причём только если состоянием s_3 из магазина M_e было вытолкнуто n раз состояние s_{no}. Попав в состояние s_{no} автомат с неизбежность попадёт в неопределённое положение при исчерпании магазина M_e. ■

Теорема 10. *Существует контекстно-свободный язык, не распознаваемый γ_{se1}-автоматом.*

Доказательство. Рассмотрим язык L, задаваемый контекстно-свободной грамматикой $G = (T = \{a, b, c, d\}, N = \{S, A, B\}, S, R)$, где $R =$

{“$S \to A$”, “$S \to B$”, “$A \to aAbAc$”, “$B \to aBbBd$”}. Язык L не принадлежит классу LL_k-языков, так как для любого наперёд заданного числа k существует достаточно длинная цепочка, принадлежащая языку L, по первым k символам которой нельзя определить, нетерминал A или нетерминал B её задаёт.

Ввиду того, что нетерминалы A и B задаются с помощью ветвящейся рекурсии, их невозможно сымитировать итеративными способами, наподобие трюка, описанного при доказательстве теоремы 9. Значит, γ_{se1}-автомат, распознающий язык L, должен использовать магазин M для обеспечения корректной вложенности нетерминалов A и B. Различить, какой из этих нетерминалов задаёт распознаваемую цепочку языка, можно только в момент считывания терминала c или d. Даже если каким-то образом изменить содержимое магазина M_e, для того чтобы отразить, какая из букв c или d была встречена, при рекурсивном возврате по магазину M, автомат сможет воспользоваться этой информацией только при возникновении неопределённой ситуации (функция переходов τ не определена для текущей конфигурации), которая невозможна ввиду её отсутствия при движении автомата по той же части пути до момента добавления этой информации в магазин M_e. ■

8. ОПТИМИЗАЦИЯ γ_1-АВТОМАТА

В данном разделе показываются конструктивные способы оптимизации, применимые к классу γ_1-автоматов, такие как минимизация состояний, устранение мнимых переходов (перехода функции переходов τ со вторым аргументом равным ϵ) и недостижимых состояний, позволяющие повысить эффективность работы автомата для практических нужд. Сначала рассмотрим устранение мнимых переходов в γ_{s1}-автоматах.

Теорема 11. *В рамках модели γ_{s1}-автомата, не сужая класс представимых им языков, можно устранить все мнимые переходы первого и второго вида.*

Доказательство. Для устранения мнимого перехода первого вида $\tau(s_1, \epsilon, m) = (s_2, m)$ достаточно заменить его множеством переходов $A \cup B \cup C \cup \{d\}$ для всех t, не участвующих в переходах из s_1, где:

- $A = \{\tau(s_1, t, m) = (s_3, m) \mid$ для каждого перехода $\tau(s_2, t, m) = (s_3, m)\}$;

- $B = \{\tau(s_1, t, m) = (s_{n+1}, ms_3 \dots s_n) \mid$ для каждого перехода $\tau(s_2, t, m) = (s_{n+1}, ms_3 \dots s_n)\}$;
- $C = \{\tau(s_1, t, s_3) = (s_3, \epsilon) \mid$ для каждого перехода $\tau(s_2, t, s_3) = (s_3, \epsilon)\}$;
- d — мнимый переход из состояния s_2, если он существует, в котором исходное состояние s_2 заменено на состояние s_1.

Мнимый переход второго вида $\tau(s_1, \epsilon, m) = (s_n, ms_2 \dots s_{n-1})$ устраняется его заменой на множество переходов $A \cup B \cup C \cup \{d\}$ для всех t, не участвующих в переходах из s_1, где:

- $A = \{\tau(s_1, t, m) = (s_{n+1}, ms_2 \dots s_{n-1}) \mid$ для каждого перехода $\tau(s_n, t, m) = (s_{n+1}, m)\}$;
- $B = \{\tau(s_1, t, m) = (s_{n+m}, ms_2 \dots s_n \dots s_{n+m-1}) \mid$ для каждого перехода $\tau(s_n, t, m) = (s_{n+m}, ms_{n+1} \dots s_{n+m-1})\}$;
- $C = \{\tau(s_1, t, m) = (s_{n-1}, ms_2 \dots s_{n-2}) \mid$ для каждого перехода $\tau(s_n, t, s_{n-1}) = (s_{n-1}, \epsilon)\}$;
- d — мнимый переход из состояния s_n, если он существует, в котором исходное состояние s_n заменено на состояние s_1.

Очевидно, что все вышеперечисленные замены не меняют язык, задаваемый γ_{s1}-автоматом. Также очевидно, что достаточно выполнить конечное число таких замен для полного устранения мнимых дуг первого и второго вида, так как каждая замена сокращает их конечное число на единицу. ■

Для устранения мнимых переходов третьего вида в модели γ_{s1}-автомата необходимо расширение перехода третьего вида до перехода $\tau(s_1, t, m_1) = (s_2, \epsilon)$, выталкивающего состояние из магазина M и использующего его для определения состояния s_2. Такое расширение ввиду роста числа переходов, зависящих от содержимого магазина, было сочтено нецелесообразным для их наглядного изображения в рамках γ-схем. В рамках класса γ_1-автомата невозможно избавится от мнимых переходов в состояния, которым сопоставлены функции f_e, изменяющие содержимое магазина M_e, так как модель γ-автомата не позволяет приписывать изменение стека M_e переходам функции переходов τ.

Некоторые из состояний γ_1-автомата могут оказаться недостижимыми из состояния s_0. Недостижимыми называются состояния, не отмеченные обходом[13] по переходам τ, τ_k и f_e (тоже рассматриваемой здесь как функция перехода) из состояния s_0. Недостижимые состояния могут быть удалены для экономии размера автомата.

[13]При обходе рассматриваются переходы, заданные синтаксисом автомата.

По γ_1-автомату часто можно построить γ_1-автомат с меньшим числом состояний, эквивалентный исходному. Соответствующий процесс называется минимизацией γ_1-автомата. Минимизация γ_1-автомата схожа с минимизацией конечного автомата [9]. Сначала все состояния автомата разбиваются на класс конечных состояний (S_{end}) и классы состояний, разбиваемые отношением эквивалентности $\equiv_0$. Пара состояний (s_1, s_2) принадлежит отношению $\equiv_0$, если верен предикат $(A_1 \vee A_2) \wedge B_1 \wedge (C_1 \vee C_2)$, где:

- терм A_1 истинен тогда и только тогда, когда из состояний s_1 и s_2 не существует мнимых переходов;
- терм A_2 истинен тогда и только тогда, когда из состояний s_1 и s_2 существуют мнимые переходы;
- терм B_1 истинен тогда и только тогда, когда состояния s_1 и s_2 имеют один вид функции f_e;
- терм C_1 истинен тогда и только тогда, когда $s_1 \in S \setminus S_k$ и $s_2 \in S \setminus S_k$;
- терм C_2 истинен тогда и только тогда, когда $s_1 \in S_k$, $s_2 \in S_k$ и $\mu(s_1) = \mu(s_2)$.

Отношение $\equiv_0$ рефлексивно и симметрично, так как все отношения термов предыдущего предиката рефлексивны и симметричны. Отношение $\equiv_0$ транзитивно, так как все отношения термов предыдущего предиката транзитивны, а отношения термов с одинаковой буквой не пересекаются. Таким образом, отношение $\equiv_0$ является отношением эквивалентности.

Так как γ_1-схемы имеют дополнительные (повышающие их наглядность) обозначения, то для минимизации γ_1-автомата в терминах γ_1-схем к предикату отношения $\equiv_0$ необходимо добавить терм A_3, проверяющий, что из вершин, соответствующих состояниям s_1 и s_2, исходят сплошные непомеченные дуги. Также необходимо добавить терм D_1, проверяющий, что вершины, соответствующие состояниям s_1 и s_2, и все переходы из них, вызванные одинаковыми пометками, имеют одинаковые пометки транслирующих процедур.

Далее каждый полученный класс эквивалентности разбивается отношением эквивалентности $\equiv_1$. Пара состояний (s_1, s_2) принадлежит отношению $\equiv_1$, если любой одинаково помеченный переход τ, τ_k и f_e (рассматриваемой здесь как функция перехода) из этих состояний ведёт к состояниям из одного класса эквивалентности предыдущего разбиения. Разбиение продолжается до тех пор, пока общее число классов

эквивалентности увеличивается, после чего совокупность представителей полученных классов эквивалентности будет состояниями минимизированного автомата.

9. ОПИСАНИЕ ТРАНСЛЯТОРА

С помощью γ_1-схем можно реализовать транслятор, осуществляющий лексический, синтаксический и семантический разборы языков программирования. Для этого к вершинам и дугам γ_1-схем добавляются пометки, состоящие из идентификаторов транслирующих процедур. Реализация процедур в формализме γ_1-схем не уточняется и может иметь общий модифицируемый контекст исполнения, позволяющий задавать трансляцию произвольной сложности.

Транслятор разделяется на графическую, текстовую и интерпретирующую части. Графическая часть описывает γ_1-схему, задающую γ_1-автомат, который распознаёт синтаксис языка. Текстовая часть содержит описание контекстно-зависимых переходов ("семантики отношений") и транслирующих процедур ("операционной семантики"), использующихся в γ_1-автомате. Интерпретирующая часть транслятора исполняет его графическую и текстовую части.

Разделение транслятора на графическую и текстовую части позволяет сочетать сильные качества обеих форм представления, что косвенно подтверждается распространёнными средствами визуального моделирования [16]. Графические спецификации подходят для проектирования и документирования из-за богатства способов представления, легче усваиваемых кратковременной памятью человека. Текстовые спецификации лучше подходят для реализации программы по причине сочетания строгости, гибкости, компактности записи и переносимости. К другим преимуществам разделения транслятора на графическую и текстовую части относятся следующие:

- простота модификаций входного языка путём наглядных изменений γ_1-схемы;
- высокая переносимость по причине интерпретируемости γ_1-автомата;
- использование γ_1-автомата для других целей, таких как тестирование;
- обособленность процедур, упрощающая их реализацию, тестирование и проверку входных и выходных условий каждой процедуры;

- эффективная раздельная трансляция γ_1-автомата и её процедур;
- общий контекст процедур более экономно расходует память за счёт отсутствия накладных расходов, присущих иерархии локальных контекстов, получаемых при вложенных функциональных вызовах;
- общий контекст процедур легче контролировать и быстрее восстанавливать в случае ошибочных ситуаций, что обычно нереализуемо в языках высокого уровня без дополнительных расходов, связанных с обработкой исключений;
- интерпретация γ_1-автомата допускает динамические оптимизации, настраивающие его на транслируемый язык или даже на конкретную программу языка в процессе её разбора;
- лёгкое изменение интерпретатора для инструментации транслятора и сбора статистики программ транслируемого языка.

Компилятор переднего плана потокового языка Sisal 3.1 системы функционального программирования [17] спроектирован с использованием рассмотренного разделения на графическую, текстовую и интерпретирующую части. Компилятор переднего плана осуществляет лексический и синтаксический разбор (совмещённый с семантическим) текста программ языка Sisal 3.1 во внутреннее представление, основанное на потоке данных. Разработанный транслятор обеспечивает простоту учёта модификаций языка, удовлетворительную скорость трансляции[14], качественные сообщения об ошибках и предупреждениях и развитые механизмы восстановления после ошибок разбора. Использование γ_1-схем позволило сократить объём текста транслятора переднего плана языка Sisal 3.1 до десяти тысяч строк по сравнению с тридцатью тысячами строк кода аналогичной части транслятора OSC 12.0 для более простой версии языка Sisal 1.2 [18].

10. ЗАКЛЮЧЕНИЕ

В статье вводится и исследуется новая автоматная модель, предназначенная для описания синтаксического разбора языков программирования. Модель допускает наглядное изображение в виде графа и упро-

[14]Невысокая скорость разбора, до десяти раз медленнее обычных показателей трансляторов промышленного уровня, объясняется тем, что задача построения высокоскоростного транслятора изначально не ставилась. В частности, высокая скорость разбора ограничивается интерпретацией γ_1-автомата и накладными расходами вызова методов COM-интерфейсов.

щает ручное построение высокоэффективных распознавателей, минуя недостатки, свойственные их автоматическому построению. В детерминированном случае введённая автоматная модель допускает класс LL_1-языков. Разбор более широких классов языков описывается неявно с помощью контекстных состояний, используемых для разбора трудных фрагментов языка и, как правило, не влияющих на наглядность графа всего автомата, что подтверждается опытом реализации компилятора переднего плана потокового языка Sisal 3.1. Показаны способы повышения эффективности автомата введённой модели, такие как минимизация состояний, устранение мнимых переходов и недостижимых состояний.

СПИСОК ЛИТЕРАТУРЫ

1. Ахо А.В., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструменты. — С.-Пб.: Вильямс, 2001. — 768 с.
2. Johnson S.C. YACC — yet another compiler compiler. — NY: Murray Hill, 1975. — 33 p. — (Tech. Rep. / AT&T Bell Labs; N 32).
3. Donnelly C., Stallman R.M. Bison Manual: Using the YACC-Compatible Parser Generator. — Boston, MA: Free Software Foundation, 2003. — 136 p.
4. Grune D., Jacobs C. Parsing techniques: a practical guide. — NJ: Ellis Horwood, 1990. — 322 p.
5. Parr T. The Complete Antlr Reference Guide. — Pragmatic Bookshelf, 2007. — 361 p.
6. Льюис Ф., Розенкранц Д., Стринз Р. Теоретические основы проектирования компиляторов. — М.: Мир, 1979. — 656 с.
7. Leermakers R. The Functional Treatment of Parsing. — Norwell, MA: Kluwer Academic Publishers, 1993. — 158 p.
8. Horspool R.N. Recursive ascent-descent parsing // Computer Languages. — NY: Pergamon Press, 1993. — Vol. 18, N 1. — P. 1–15.
9. Полетаева И.А. Методы трансляции: Конспект лекций. — Новосибирск: НГТУ, 1997. — 59 с.
10. Nunes-Harwitt A. CPS Recursive Ascent Parsing // Proc. of Internat. LISP Conf., 2003. — 6 p.
11. McDonald J. Interactive Pushdown Automata Animation // ACM SIGCSE Bull. NY: ACM Press, 2002. — Vol. 34, N 1. — P. 376–380.
12. Cavalcante R., Finley T., Rodger S.H. A visual and interactive automata theory course with JFLAP 4.0 // ACM SIGCSE Bull. — NY: ACM Press, 2004. — Vol. 36, N 1. P. 140–144.
13. Вирт Н., Йенсен К. Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1989. — 255 с.

14. Касьянов В.Н., Поттосин И.В. Методы построения трансляторов. — Новосибирск: Наука, 1986. — 330 с.
15. Стасенко А.П. Графический метаязык для описания транслятора // Сборник трудов аспирантов и молодых учёных "Молодая информатика". — Новосибирск: ИСИ СО РАН, 2005. — С. 105–113.
16. Кузнецов Б.П. Психология автоматного программирования // Журнал BYTE, 2000. — № 11. — С. 22–29.
17. SFP — An interactive visual environment for supporting of functional programming and supercomputing / Kasyanov V.N., Stasenko A.P., Gluhankov M.P., Dortman P.A., Pyjov K.A., Sinyakov A.I. // WSEAS Transactions on Computers. — Athens: WSEAS Press, 2006. — Vol. 5, N 9. — P. 2063–2070.
18. Стасенко А.П. Использование автоматного подхода для построения компилятора переднего плана // Тез. конф.-конкурса "Технологии Microsoft в теориии и практике программирования". — Новосибирск, 2006. — С. 37–39.

С. В. Филябин

ТЕХНОЛОГИЯ АВТОМАТИЗАЦИИ МОНИТОРИНГА И КОНТРОЛЯ ЛЕГАЛЬНОСТИ ФИНАНСОВЫХ ОПЕРАЦИЙ СОВРЕМЕННЫХ КРЕДИТНЫХ ОРГАНИЗАЦИЙ

ВВЕДЕНИЕ

В последние годы существенно возросла необходимость контроля финансовых операций в кредитных организациях, предоставляющих различные сервисы: кредитование, переводы денежных средств, оплата услуг, управление счетами через Интернет. Наличие и многообразие сервисов дает большие преимущества при осуществлении финансовых операций, но минусом является отсутствие продуманных и четких механизмов мониторинга информационных потоков с целью недопущения легализации доходов, полученных преступным путем, финансирования экстремистской деятельности, контроля оттока средств за пределы РФ.

Вопросы обеспечения информационной безопасности для современного банка являются жизненно важными по ряду причин:

- основным операционным материалом банка являются деньги. С точки зрения информационной безопасности это существенно повышает риски;
- автоматизированные банковские системы (АБС) в большинстве случаев гетерогенны, то есть представляют собой большой набор бэк-офисных и фронт-офисных программных систем от различных производителей. Управление этими системами осложняется территориальной распределенностью, наличием филиалов и офисов. Очевидно, что низкая степень интеграции и незаконное манипулирование информационными потоками между подсистемами могут привести к серьезным убыткам;
- банк — это точка пересечения внутренних корпоративных сетей, публичных сетей (Интернет) и коммерческих финансовых сетей (Western Union, Visa, Master Card, SWIFT). Наличие удаленного доступа к информационной системе банка посредством этих сетей может привести к несанкционированным финансовым операциям;
- банк хранит конфиденциальную информацию своих клиентов, которая является коммерческой тайной организаций. Возможна утеч-

ка персональных данных о клиентах, физических лицах, и в результате — проведение мошеннических операций.

Основной упор при защите банковских информационных систем необходимо делать на превентивные меры, анализ рисков, построение процессов обеспечения непрерывности деятельности и управление инцидентами.

Требования к защите автоматизированных информационных систем современных банков:

- комплексность — учет всех направлений деятельности банка и возможных информационных атак по каждой из подсистем, создание корпоративной политики безопасности;
- интегрируемость — создание единых интерфейсов обмена информационными сообщениями между подсистемами, использование современных интеграционных решений и платформ;
- легитимность — соответствие положениям и указаниями центрального банка (ЦБ) РФ и других регулирующих органов, международным стандартам ISO 13569 «Banking and related financial services — Information security guidelines», Payment Card Industry Data Security Standard (PCI DSS) и другим;
- управляемость — возможность построения эффективной системы бизнес-процессов банка на основе АБС, оперативный контроль информационных потоков;
- масштабируемость — возможность вносить изменения в архитектуру АБС с целью оптимизации работы набора подсистем;
- отказоустойчивость — возможность быстрого и полного восстановления работоспособности АБС при сбоях.

1. ЗАДАЧА ПРОЕКТА

Подробнее остановимся на легитимности осуществляемых операций. Центральный банк РФ в 2003 году установил порядок представления кредитными организациями в уполномоченный орган сведений об операциях с денежными средствами или иным имуществом, подлежащих обязательному контролю, а также иных операциях с денежными средствами или иным имуществом, связанных с легализацией (отмыванием) доходов, полученных преступным путем (Положение 207-П). В положении определяются основные критерии подозрительности и принципы анализа финансовых операций, вводится перечень официальных справочников экстремистских организаций и физических лиц, справочник государств, не входящих в группу

стран, противодействующих экстремизму, определяется формат электронной отчетности и правила предоставления отчетности.

В случаях, когда службе безопасности и бизнес-подразделениям банков необходимы дополнительные отчеты по легальности движения средств, вырабатываются внутренние регламенты анализа операций, данных и документов, определяются внутренние критерии их подозрительности.

Основной задачей разрабатываемого проекта является создание настраиваемой системы мониторинга и анализа финансовых операций по критериям подозрительности с учетом законодательства РФ (207-П) и внутренних регламентов банков. Основными требованиями к системе являются возможность интеграции с многоплатформенными АБС, масштабируемость, настраиваемый набор критериев подозрительности, наличие инструментальных средств для доработки и получения отчетов по подозрительным операциям.

2. ИНТЕГРАЦИОННАЯ ПЛАТФОРМА

Информационная структура любой финансовой организации — это хранилище данных, то есть совокупность специализированных баз данных (БД) для различных подсистем бизнеса. Каждая база данных имеет собственную структуру.

Технология хранилищ данных позволяет эффективно решать следующие задачи:

- интеграция бизнес-данных всех филиалов и подразделений;
- автоматизация технологий управления;
- автоматизация работ по выпуску консолидированной отчетности;
- аудит филиалов и дочерних предприятий;
- обеспечение архивными данными для оценки развития бизнеса;
- построение единого информационного пространства распределенной организации;
- предоставляет механизм объединения БД бизнес-подсистем в единое целое, используя интерфейс составных запросов к нескольким БД.

На современном рынке программного обеспечения существует большой выбор интеграционных платформ. Для реализуемого проекта был выбран набор программных продуктов IBM WebSphere. Он позволяет создать платформу для электронного бизнеса, основанную на использовании возможностей Интернет. Программная платформа WebSphere базируется на

широко распространенных стандартах, таких как Java, XML, J2EE. Это позволяет легко интегрировать разнотипные ИТ-среды, оперативно адаптироваться к изменению задач бизнеса, упростить доступ внешних пользователей к ресурсам системы, что приводит к увеличению производительности, уменьшению издержек и к более быстрому выходу на рынок с новыми продуктами и услугами.

Платформа WebSphere формирует фундамент законченной оболочки бизнес-решений. Большое количество различных продуктов обеспечивает решение многих проблем, с которыми сталкиваются предприятия любых размеров. IBM WebSphere предоставляет единую точку доступа к приложениям, контенту, процессам и пользователям в рамках интегрируемых информационных систем.

3. АРХИТЕКТУРА СИСТЕМЫ

С учетом описанных ранее ограничений и недостатков современных банковских информационных систем, требований законодательства и бизнеса по контролю легальности операций, а также с учетом технологических возможностей платформы IBM WebSphere, оптимальной представляется архитектура АБС, показанная на рис.1. В данном случае подсистема анализа финансовых операций представляет собой встраиваемый в информационную структуру организации компонент, использующий механизм составных запросов-сообщений через IBM WebSphere и предоставляющий интерфейс к своим внутренним функциям проверки.

Модуль анализа операций состоит из следующих блоков: блок настройки, блок анализа структурированной информации, блок анализа неструктурированной информации, блок анализа повторяющихся операций, блок отчетности. Схема модуля показана на рис. 2.

Блок настройки позволяет указывать, какие сущности (суммы, даты, наименования, текстовые поля) необходимо контролировать, в каких БД и таблицах, позволяет указывать тип проверки: «структурированная-неструктурированная-повторяющиеся операции», и в зависимости от типа проверки определять критерии подозрительности.

Рис. 1. Архитектура АБС. Интеграция системы анализа финансовых операций

Блок анализа структурированной информации анализирует значения конкретных финансовых реквизитов (суммы, даты, количество и лимиты операций). Например, если указать контролируемый реквизит — сумму документа, критерий подозрительности — превышение определенной суммы, то все документы системы, где сумма превышает заданную, будут считаться подозрительными.

Блок анализа неструктурированной информации использует следующие предметно-ориентированные компоненты и справочники для анализа текстов на естественном языке: компонент приведения слов к инициальным формам Dialing, компонент поиска совпадений, официальный справочник лиц-экстремистов, официальный справочник экстремистских организаций, справочник фраз, считаемых подозрительными (пример фразы: «содействует экстремизму»), справочники синонимов, антонимов, паронимов, конверсивов.

Система Dialing выбрана в качестве компоненты приведения слов к инициальными формам по следующим причинам:

- Dialing является разработкой, над которой в период с 1999 по 2002 год работала группа ведущих российских учёных-лингвистов и программистов;
- система объединяет в себе достоинства других известных систем, таких как ФРАП (система французко-русского автоматического перевода) [5], ПОЛИТЕКСТ (система анализа политических текстов) [6], Микрокосмос [7] и других;
- Dialing включает в себя модули работы с русским и английским языками;
- базовая функциональность системы реализована в рамках технологии COM, что даёт возможность использования её во внешних приложениях.

Инициальной или начальной формой считается: для имен существительных — именительный падеж, единственное число; глаголов — форма инфинитива; для прилагательных — именительный падеж, единственное число, мужской род.

При поиске подозрительных наименований используются следующие критерии:

- инициальная форма наименования присутствует в справочниках экстремистов;
- установлено, что анализируемый объект имел и/или имеет финансовые взаимодействия с экстремистами;
- наименования присутствует в массиве прошлых наименований подозрительного объекта.

При поиске подозрительных текстовых фраз в базах клиентов и документах системы используются следующие критерии:

- фраза подозрительна, если совпали нескольких подряд идущих букв в проверяемой и подозрительной фразах:
- фраза подозрительна, если все слова подозрительной фразы входят в проверяемую фразу:
- фраза подозрительна, если ее подфразой является подозрительная фраза
- фразы подозрительна, если совпали 50% букв проверяемой и подозрительной фразы без учета их последовательности и отдаленности друг от друга;
- фраза подозрительна, если часть проверяемой фразы имеет синоним, подстановка которого делает фразу подозрительной.

Опишем некоторые классы поисковых алгоритмов. Классы можно условно разделить на алгоритмы прямого поиска и алгоритмы, требующие

предварительной обработки документов, создания вспомогательного индексного файла, для ускорения и упрощения поиска.

Алгоритмы прямого поиска — поиск происходит при помощи последовательного просмотра документов.

Достоинства метода: неограниченные возможности по приближенному и нечеткому поиску. Прямой поиск работает непосредственно по оригинальным документам без искажений, любое индексирование всегда связано с упрощением и нормализацией терминов, т. е. с потерей информации. Несмотря на неоптимальность метода, последние несколько десятилетий прямой поиск интенсивно развивается. Выдвинуто множество идей, сокращающих время поиска в несколько раз. Алгоритмы прямого поиска часто разбиваются на отдельные составляющие, которые при группировке дают хорошие результаты. Существующие алгоритмы непрерывно оптимизируются.

Прямой просмотр всех текстов — медленный процесс, но алгоритмы данного класса с успехом применяются в Интернете. Примером может послужить поисковая система Fast Search использующая чип, реализующий логику прямого поиска упрощенных регулярных выражений и кластер из 256 чипов на одной плате. Такая архитектура позволяет системе обслуживать большое количество запросов в единицу времени.

Так же существует множество программ, объединяющих разнотипные классы алгоритмов, например, индексный поиск с дальнейшим прямым поиском внутри блока.

Класс алгоритмов с использованием инвертированных файлов

Инвертированный файл — специальная структура данных. В литературе аналог такой структуры называется «конкордансом» — алфавитно-упорядоченный исчерпывающий список слов из одного текста или принадлежащих одному автору (например, конкордансы произведений Владимира Даля, словарь-конкорданс публицистики Ф. М. Достоевского).

В терминах программирования баз данных инвертированный файл получают при построении индекса таблицы по ключевому полю. После получения упорядоченного по алфавиту списка слов, где для каждого слова перечислены все адреса-позиции, в которых это слово встречается, алгоритм отыскивает нужное слово и возвращает информацию по позициям и встречаемости данного слова в исследуемом тексте.

В инвертированном файле можно хранить информацию не только об адресе, но и другие атрибуты: номер слова, различные гипертекстовые теги

слова. В классической теории информационного поиска (Information Retrieval) в инвертированном файле хранят номер документа и число употреблений этого слова в нем.

Для оптимизации использования дискового пространства используются алгоритмы сжатия информации инвертированных файлов: LZW, алгоритм Хафмана и другие. Необходимо учитывать, что при архивировании файла возрастает нагрузка на процессор при его упаковке-распаковке.

В нашем случае при поиске подозрительных операций и документов можно использовать справочник конкордансов, составленный по досье клиентов. В качестве слов данного справочника будут выступать слова и фразы, считающиеся подозрительными, а адресами будут идентификационные номера клиентов, в досье которых встречается данное слово или фраза. Там же будет храниться и частота встречаемости данной фразы в досье. Справочник будет регулярно дополняться новыми данными в фоновом режиме.

Математические модели

При повышенных требованиях к качеству поиска и большом объеме информации, возможно построение математической модели поиска — на основании модели выводится формула, позволяющая системе принять решение: какой документ считать найденным и как его ранжировать. Модели традиционного информационного поиска можно разделить на три вида: теоретико-множественные (булевская, нечетких множеств, расширенная булевская), алгебраические (векторная, обобщенная векторная, латентно-семантическая, нейросетевая) и вероятностные.

Булевские модели — простейший пример, если слово встречается в документе, то результат функции: true, иначе false.

Ранжирование в векторной модели основано на естественном статистическом наблюдении: чем больше локальная частота термина в документе и больше «редкость» — обратная встречаемость в документах термина, — тем выше вес данного документа по отношению к термину.

Вероятностная модель: вероятность оказаться релевантным для каждого следующего документа рассчитывается на основании соотношения встречаемости терминов в релевантном наборе и в остальной, нерелевантной части коллекции. Вероятностные модели располагают документы в порядке убывания «вероятности оказаться релевантным».

Лингвистические алгоритмы

Лингвистическими считаются методы, в основе своей использующие различные лингвистические правила, справочники и словари (морфологические, синтаксические, семантические), созданные учеными-лингвистами, например, словари синонимов, антонимов и др. Основная масса языков требует должного уровня лингвистической обработки по причине наличия в языке падежей, склонений, множественного и единственного числа, времен глаголов и др. Список задач класса лингвистических алгоритмов:

- автоматическое определение языка документа;
- токенизация (графематический анализ): выделение слов, границ предложений;
- исключение неинформативных слов;
- лемматизация : приведение слов к инициальным формам;
- разделение сложных слов для некоторых языков.

Блок анализа повторяющихся операций согласно критериям, указанным в блоке настроек, например, количество документов с одинаковой корреспонденцией и их общая сумма за определенный период, количество зачислений на один счет, на счета одного клиента и их общая сумма, количество списаний с одного счета, со счетов одного клиента и их общая сумма, модуль анализа помечает подозрительные операции и отправляет их на дополнительный контроль.

Рис. 2. Общая схема построения модуля анализа операций

Блок отчетности использует данные, собранные в результате проверок всех типов (операции и причины подозрительности), и отображает их в установленном виде для последующего анализа уполномоченным сотрудником. Позволяет выгружать файлы в нужном формате для отправки в ЦБ.

ЗАКЛЮЧЕНИЕ

Хочется отметить, что предлагаемое решение имеет несколько путей развития и доработки: в частности, планируется определение оптимального набора контекстных поисковых алгоритмов различных классов (прямой поиск, индексные, лингвистические), подключение иностранных словарей для анализа текстовой информации на других языках, использование справочников буквосочетаний для разных языков с целью идентификации наименований, применение правил транскрипции, расширение критериев подозрительности и возможностей настройки анализатора.

СПИСОК ЛИТЕРАТУРЫ

1. Построение хранилищ данных // Банковские технологии 7-8/2002. — М.: ПрофиПресс, 2002. — С. 56–60.
2. Лукацкий А. Информационная безопасность банка. — http://www.cisco.com/global/RU/news/media/0209.shtml
3. Анализ уровня информационной безопасности банка. — http://kiev-security.org.ua/box/9/7.shtml
4. Селезнев К. Обработка текстов на естественном языке.— http://www.osp.ru/os/2003/12/048.htm
5. Леонтьева Н.Н. Система французско-русского автоматического перевода (ФРАП): лингвистические решения, состав, реализация // МП и ПЛ. Проблемы создания системы автом. перевода / Сб. научн. тр. МГПИИЯ им. М. Тореза. — М., 1987. — Вып. 271. — С. 6–25.
6. Леонтьева Н.Н. ПОЛИТекст: информационный анализ политических текстов // Сб. НТИ. — 1995. — Сер. 2, № 4.
7. Raskin V., Nirenburg S. Lexical Semantics of Adjectives // Recent Papers from the Mikrokosmos and Corelli Projects. — New Mexico State Univ., 1996. — Vol. 2.

М.В. Шпак

О НЕКОТОРЫХ МЕТОДАХ МОДЕЛИРОВАНИЯ АППАРАТУРЫ ЭЛЕКТРОМАГНИТНОГО КАРОТАЖА

При поиске, разведке и эксплуатации полезных ископаемых традиционными и одними из самых популярных являются методы каротажа с использованием электромагнитного поля. Их условно можно разделить на 3 типа:

- электрический на постоянном токе;
- низкочастотный индукционный;
- высокочастотный индукционный.

К первому типу относятся приборы бокового каротажа, многозондовые приборы электрического каротажа. Второй тип представлен различными вариациями приборов низкочастотного индукционного каротажа (частота — десятки кГц); ярким представителем третьего типа является ВИКИЗ.

Следует отметить, что на сегодняшний день сервисные компании зачастую пользуются «сырыми» данными со скважин без соответствующей обработки. Объясняется это тем, что создание алгоритмов интерпретации сопряжено с большими техническими и вычислительными сложностями.

Мы будем рассматривать первых 2 метода в силу того, что для ВИКИЗа на сегодняшний день создано достаточно много интерпретационных комплексов, а для первых двух методов алгоритмы имеют множество недостатков либо являются коммерческой тайной сервисных компаний.

Интерпретатора данных электромагнитного каротажа интересуют 3 параметра разреза: удельное электрическое сопротивление (УЭС) неизменной части пласта (Rp), УЭС зоны проникновения скважной жидкости (Rzp) и глубина проникновения, а точнее, отношение диаметра зоны проникновения к диаметру скважины (D/d). Таким образом, алгоритмы интерпретации должны решать обратную задачу электрического (в случае электрического каротажа на постоянном токе) и обратную задачу индукционного каротажа.

Алгоритмы интерпретации для них будут разными вследствие двух важных факторов:

1. Электрический каротаж является контактным методом, индукционный — нет.
2. При решении задачи индукционного каротажа можно использовать аналитические и полуаналитические методы, электрического — нет.

Рассмотрим эти методы подробнее.

ЭЛЕКТРИЧЕСКИЙ КАРОТАЖ НА ПОСТОЯННОМ ТОКЕ

В случае постоянных по времени полей и отсутствия объемных источников электростатический потенциал в расчетной области Ω определяется уравнением

$$\varphi\big|_{\Gamma_i^-} = \varphi\big|_{\Gamma_i^+} + \Delta\varphi, \tag{1}$$

При наличии дипольных горизонтальных источников в осесимметричных средах с изотропной проводимостью σ в цилиндрических координатах оператор L имеет вид

$$L\varphi = -\frac{1}{r}\frac{\partial}{\partial r}(\sigma r\frac{\partial\varphi}{\partial r}) - \frac{\partial}{\partial z}(\sigma r\frac{\partial\varphi}{\partial z}) - \frac{1}{r}\frac{\partial}{\partial\phi}(\frac{\sigma}{r}\frac{\partial\varphi}{\partial r})$$

Характерной для геофизических приложений является краевая задача, в которой требуется найти значения потенциалов V_k на электродах S_k при заданных величинах токов

$$Ik = \int_{S_a} \sigma\frac{\partial u}{\partial n}ds,\ k = 1,\ldots,N_e \tag{2}$$

При этом на внутренних границах раздела сред Γ_i должны выполняться условия сопряжения с возможными скачками потенциала и нормальной составляющей вектора напряженности $\vec{E} = -\sigma\nabla\varphi$

$$\varphi\big|_{\Gamma_i^-} = \varphi\big|_{\Gamma_i^+} + \Delta\varphi,\quad (\vec{E},\vec{n})\big|_{\Gamma_i^-} = (\vec{E},\vec{n})\big|_{\Gamma_i^+} + \Delta E \tag{3}$$

На внешних границах расчетной области из физических соображений ставятся краевые условия первого, второго или третьего рода:

$$\varphi\big|_{\Gamma 1} = g1,\ \frac{\partial\varphi}{\partial n}\bigg|_{\Gamma 2} = g2,\ \varphi + \gamma\frac{\partial\varphi}{\partial n}\bigg|_{\Gamma 3} = g3 \tag{4}$$

Полное решение такой задачи (1)–(4) представляется в виде

$$u = u^{(0)} + \sum_{k=1}^{Ne} u^{(k)}V_k \tag{5}$$

где $u^{(0)}$ удовлетворяет однородным условиям на электродах ($u^{(0)}|s_k = 0$) и (3)–(4) на остальной Γ, а каждое частное решение $u^{(k)}$: $u^{(k)}|s_{k'} = \delta_{k,k'}$, $k' = 1, \ldots, N_e$ и подчиняется однородным условиям (3) и (4) на Γ.

Решение такой задачи возможно либо реализацией различных методов решения прямой задачи электрического каротажа (например, описанной в [1]), либо с помощью различных пакетов с использованием методов конечных элементов (например Comsol Multiphysics).

Для решения обратной задачи электрического каротажа возможно использование различных методов, в том числе таких, как поиск решения в вертикальной плоскости в виде полинома, использование данных с других приборов для получения начального приближения и для исключения некоторых неизвестных параметров. Одним из перспективных методов является итерационный алгоритм, основанный на теории возмущений. Его описание дается в [1], математическое обоснование метода — в [2]. Одним из ключевых моментов такой методики является использование большого количества однопластовых палеток. Одной из основных проблем реализации этого метода является скорость решения прямой задачи, поэтому для эффективной работы комплекса в целом ключевым вопросом стоит создание быстрого алгоритма решения прямой задачи.

НИЗКОЧАСТОТНЫЙ ИНДУКЦИОННЫЙ КАРОТАЖ

В общем случае прибор индукционного каротажа состоит из излучателя и системы приемных катушек. Т.к. размеры прибора намного превосходят размеры излучающих и приемных катушек, их можно рассматривать как одиночные круговые витки, характеризующиеся своей площадью S. Будем рассматривать только приборы, имеющие осевую симметрию, т.е. такие, где оси излучающего витка и всех приемных витков совпадают. Тогда в однородной среде (вообще в среде, имеющей симметрию относительно оси прибора) электромагнитное поле обладает осевой симметрией, вектор-потенциал $\vec{A}$ имеет только компоненту вдоль φ (цилиндрические координаты (r,φ,z)), которую в дальнейшем будем обозначать просто A, и для которой можно записать цилиндрическое уравнение Гельмгольца:

$$\Delta_2 A - \frac{A}{r^2} + k^2 A = -\mu_0 \mu j_0 \qquad (6)$$

где $k^2 = \varepsilon_0 \mu_0 \varepsilon \mu \omega^2 + i \mu_0 \mu \sigma \omega$, ε_0 и μ_0 — диэлектрическая и магнитная константы вакуума, ε и μ — относительные диэлектрическая и магнитная проницаемости окружающей прибор среды, σ — проводимость этой среды, ω — частота возбуждающего электромагнитное поле тока и j_0 — плотность распределения этого тока.

Итоговым результатом ПО интерпретации должно быть распределение проводимости в пространстве с осевой симметрией.

Используя приближенное выражение функции Грина (устремив размеры излучающих и приемных колец к нулю) для уравнения (6) итоговая проводимость среды дается выражением:

$$\sigma_k = \frac{2\pi}{S} \cdot \frac{L}{1-2^{-2/3}} \cdot \int \left[\frac{1}{\left(R_0^{(1)}\right)^3} - \frac{1}{2} \frac{1}{\left(R_0^{(2)}\right)^3} \right] G\left(x', x_o\right) r'^2 \, \sigma\left(x'\right) dx' \qquad (7)$$

где $R_0^{(1,2)} = \left| x_{\text{г}}^{(1,2)} - x' \right|$ — расстояния от приемных витков до исследуемой точки, а G — функция Грина уравнения (6). Ядро интегрального оператора вместе с коэффициентом, стоящим перед интегралом, называют функцией отклика. Ее значение показывает, какая часть сигнала набирается в точке x' от проводимости $\sigma\left(x'\right)$.

Функция G и, соответственно, функция отклика f для произвольного распределения проводимости, к сожалению, неизвестна. Однако, в первом, так называемом Борновском приближении, когда в среднем проводимость среды мала, и индуцированные токи слабо влияют на распределение электромагнитного поля, можно считать, что $G = G^{\sigma=0}$. В этом случае функция отклика дается выражением:

$$f = \frac{L}{2} \cdot \frac{1}{1-2^{-2/3}} \cdot \frac{\left(R_0^{(2)}\right)^3 - \left(R_0^{(1)}\right)^3 / 2}{\left(R_0^{(1)} R_0^{(2)} R_0\right)^3} r'^3 \qquad (8)$$

Функции отклика далеки от нуля в достаточно большой области и принимают как положительные, так и отрицательные значения. Таким образом, измеряемая зондами величина σ_k является усредненной по большой площади и неудобна для интерпретации распределения проводимости в среде.

Один из методов построения решения обратной задачи основан на так называемых синтезированных зондах, описанных в работе [3]. Там дается представление о том, каким образом локализовать показания искусственно полученных зондов для дальнейшей интерпретации. Линейным преобразованием, которое мы можем произвести с функциями $\sigma_k^{(n)}$, является преобразование вида:

$$\tilde{\sigma}_k^{(m)}\left(z\right) = \sum_{n=1}^{N} \int w_{(n)}^{(m)}\left(z'\right) \sigma_k^{(n)}\left(z - z'\right) dz', \; m = 1 \ldots M \qquad (9)$$

При этом веса $w_{(n)}^{(m)}$ подбираются согласно требуемому виду целевой функции.

На основе этого набора показаний можно построить решение обратной задачи. Эта возможность достигается путем решения следующей задачи:

$$\tilde{\sigma}^{(m)}(z) = f(Rp, Rzp, D/d),$$

функция f линейна по первым двум аргументам и нелинейна по третьему. Причем само D / d в общем случае зависит от Rp. Однако можно подобрать такие параметры преобразования (9), что в большом диапазоне Rp этой зависимостью можно пренебречь. Таким образом, мы получаем задачу о решении системы уравнений, линейной по двум и нелинейной по одному аргументу, решаемой численно.

При каротаже часто возникает необходимость в оценке параметров разреза «на лету». С этой целью разработан механизм определения параметров разреза, основанный на итерационном процессе выборки Rp, Rzp и D / d, удовлетворяющих показаниям зондов. Массив для выборки получен полуаналитическими методами, описанными в [4].

Поскольку в этом методе используется поточечный расчет, в дальнейшем возможна доработка метода с учетом разрешения каждого из зондов и частичной локализации по глубине. При этом мы приходим ко второму методу обратной задачи, результаты решения которой при тестировании на экспериментальных данных показывают удовлетворительную сходимость решения и восстановление параметров разреза. Однако при экстремальных значениях параметров скважины сходимость метода неудовлетворительная и требует дополнительно некоторых корректировок решения, таких как введения нормы при нахождении минимума разницы в показаниях зондов, расширение палеток на частные случаи.

В силу того, что для программ интерпретации требуется достаточно большие временные и аппаратные ресурсы, одним из возможных направлений дальнейшей деятельности видится создание более эффективных алгоритмов с использованием аналитических и полуаналитических методов.

СПИСОК ЛИТЕРАТУРЫ

1. Друскин В.Л. Разработка методов интерпретации бокового каротажного зондирования в однородных осесимметричных средах, канд. дисс. / МГУ. М., 1984.
2. Друскин В.Л., Книжнерман Л.А. Об одном итерационном алгоритме решения двумерной обратной задачи электрокаротажа // Геология и геофизика. — 1987. — № 9. — С. 118–123.
3. Зимовец С.В., Шпак М.В. Программное обеспечение интерпретации прибора индукционного каротажа. Технологии Microsoft в теории и практике програм-

мирования // Тез. докл. Конф.-конкурса работ студентов, аспирантов и молодых ученых. — Новосибирск, 2007. — С. 194.

4. Кауфман А.А. Введение в теорию геофизических методов. Ч. 2. Электромагнитные поля. — М.: Недра, 2000. — 483 с.

СОДЕРЖАНИЕ

CONTENTS

УДК 519.68 + 681.3.06

Экспериментальное исследование новой стратегии тестирования / Арапбаев Р.Н. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 7–23.

В данной работе проведено экспериментальное сравнение результатов наиболее известных алгоритмов анализа зависимостей по данным, таких как новая стратегия тестирования, Эпсилон-тест и алгоритм Майдана. Эксперимент проведен с использованием инструмента Petit, разработанного в Мэрилендском университете как расширенный вариант инструмента tiny и с использованием системы SUIF, разработанной в Стенфордском университете. Для эксперимента использованы два вида данных. Первый вид — набор тестовых научных программ NASA и PERFECT Club benchmarks, где каждая программа включает от 500 до 18000 строк. Второй вид — набор из 16 циклов, собранный из работ, аналогичных нашей. — Библиогр.: 23 назв.

Experimental research of new strategy of testing / Arapbaev R.N. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 7–23.

In this paper we present an experimental evaluation of several data dependence algorithms, including the new strategy , Epsilon-test and algorithm of Maydan. We compare these algorithms in terms of accuracy and efficiency. We conducted our experiments using the Petit Tool V1.1, developed at the University of Maryland as an extension to the Tiny Tool and using system SUIF, developed at the University of Stanford. Two types of data are used for experiment. The first type is a set of test scientific programs of NASA and PERFECT Club benchmarks, where each program consists of 500 to 18000 lines. The second type is a set of 16 cycles collected from papers similar to ours. — Refs: 23 titles.

УДК 519.68 + 681.3.06

Динамический распределенный ПН-алгоритм для раскраски w-совершенных графов / Евстигнеев В.А., Турсунбай кызы Ы. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 24–30.

Данная работа посвящена раскраске w-совершенных графов в рамках распределенной модели вычислений, которая использует широко известную стратегию ПН-алгоритма. Класс w-совершенных графов довольно широкий и содержит в себе такие практически интересные классы графов, как, например, класс хордальных графов, который является одним из наиболее изученных и широко применяемых классов графов. — Библиогр.: 12 назв.

The dynamic distributed SL-Algorithm for coloring w-perfect graphs / Evstigneev V.A., Tursunbay kyzy Y. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 24–30.

The given work is devoted to coloring w-perfect graphs within the framework of the distributed model of calculations, which uses a well-known strategy of SL-algorithm. The w-perfect graphs class is rather wide and comprises such practically interesting graph classes, as for example, the chordal graph class, which is one of the most actively investigated and widely used graph classes. — Refs: 12 titles.

УДК 519.68 + 681.3.06

Временная развёртка внутреннего представления IR2 языка Sisal 3.1 / Идрисов Р.И. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 31–37.

Язык Sisal реализует потоковую модель вычислений и является одним из самых известных языков такого типа. Он также позиционируется как замена языка Fortran для вычислений, поскольку в отличие от других потоковых языков имеет синтаксис, более схожий с привычными языками программирования, такими как Pascal. Потоковая организация вычислений обеспечивает более естественное распараллеливание кода. Механизм однократного присваивания сильно упрощает анализ зависимостей.

Целью данной работы является формулирование задачи распараллеливания в терминах внутреннего представления компилятора Sisal 3.1 IR2. — Библиогр.: 3 назв.

Time tracing for SISAL 3.1 internal representation IR2 / Idrisov R.I. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 31–37.

SISAL is the one of the better known languages with a flow model of computation. It has a Pascal-native syntax and is positioned as a FORTRAN replacement in computation programming. The data flow computation model brings a more natural code parallelization. The single assignment mechanism simplifies data dependency analysis.

The aim of this paper is formulating the program parallelization task in terms of the SISAL 3.1 compiler internal representation IR2. — Refs: 3 titles.

УДК 519.68 + 681.3.06

Методы межпроцедурного анализа / Идрисов Р.И. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 38–55.

Межпроцедурный анализ на сегодняшний день является неотъемлемой частью оптимизирующего компилятора. Смысл анализа заключается в том, чтобы определить воздействие процедурного вызова на контекст вызова и воздействие контекста на алгоритм вызываемой процедуры. В зависимости от задачи некоторые части анализа могут быть упрощены или опущены. К примеру, для оптимизатора последовательного кода информация о совмещениях (псевдонимах) может оказаться несущественной. В автоматических распараллеливающих системах находят применение все методики межпроцедурного анализа. Межпроцедурный анализ применяется также в системах разработки программного обеспечения для контролирования системы типов в больших проектах. Исключение межпроцедурного анализа из компилятора возможно, когда оптимизирующие алгоритмы не используют информацию о межпроцедурном потоке данных. В этом случае класс возможных оптимизаций сильно сужается. Можно также исключить межпроцедурный анализ за счёт осуществления прямой подстановки кода на место вызова, но это ведёт к экспоненциальному увеличению кода и не всегда является возможным.

В работе рассмотрены основные методы межпроцедурного анализа с ориентацией на автоматическую распараллеливающую систему. — Библиогр.: 15 назв.

Methods of interprocedural analysis / Idrisov R.I. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 38–55.

Interprocedural analysis is an inalienable part of modern optimizing compiler. Such analysis gives the compiler facts about the environmental changes caused by procedure call and the execution context of the procedure. Some parts of interprocedural analysis may be simplified or even eliminated according to the optimizations used. For example, alias analysis may be insignificant for sequential code compiler. Automatic parallelization systems require more precise interprocedural information for loop parallelization and interprocedural optimizations. In automatic parallelization system any information may increase parallelism. Precise interprocedural alias information is required in the software engineering systems for type control systems. It is possible to exclude interprocedural analysis when optimization algorithms don't use any information on interprocedural data flow; it reduces the set of possible optimizations. Procedure boundaries can be eliminated by replacing each procedure call by a copy of the called procedure; it makes the code exponential larger which is undesired and not always possible.

This paper observes main the interprocedural analysis methods from the point of an automatic parallelization system. — Refs: 15 titles.

УДК 519.68 + 681.3.06

Язык программирования Sisal 3.2 / Касьянов В.Н., Стасенко А.П. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 56–134.

В статье описывается синтаксис и семантика новой версии входного языка Sisal 3.2 системы параллельного программирования SFP, разрабатываемой в Институте систем информатики имени А.П. Ершова СО РАН. Язык Sisal 3.2 является языком функционального программирования, ориентированным на написание потоковых программ для научных вычислений. К основным нововведениям в языке Sisal 3.2 относительно языка Sisal 3.1 относится поддержка многомерных массивов, пользовательских типов с параметрами, обобщенных процедур, инородных типов и процедур. — Библиогр.: 17 назв.

Sisal 3.2 programming language / Kasyanov V.N., Stasenko A.P. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 56–134.

The paper describes the syntax and semantics of a new version of the Sisal 3.2 programming language designed to write data-flow programs for scientific computations. The main features of Sisal 3.2 language (as compared to Sisal 3.1 language) include the support of multidimensional arrays, parametric user types, generalized procedures, foreign types and procedures. Sisal 3.2 is a source language of the system of functional programming (SFP) which is currently being developed in the A.P.Ershov Institute of Informatics Systems. — Refs: 17 titles.

УДК 519.68 + 681.3.06

Вейвлет-обработка данных в геофизических исследованиях скважин / Крайниковский С.С. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 135–143.

В статье даётся обзор геофизических исследований скважин и методов обработки данных. Основным содержанием статьи является задача фильтрации каротажных данных и расстановки границ (выделения пластов). Оригинальное использование вейвлет-преобразований даёт результаты, которые могут быть лучше, чем те, которые получены традиционными методами расста-

новки границ при обработке геофизических данных. — Библиогр.: 5 назв.

Wavelet data processing in geophysical investigation in boreholes / Krainikovsky S.S. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 135–143.

The article presents an overview of geophysical investigation in boreholes and data processing methods. The problem of filtering logging data and setting borders (determination of layers) is the main subject of the article. The original usage of wavelet transformations gives some results that may be better than the results obtained by traditional methods of setting borders in geophysical data processing. — Refs: 5 titles.

УДК 519.68 + 681.3.06

Разработка графических интерфейсов и визуализация данных в геофизических программных системах / Крайниковский С.С. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 144–149.

В статье содержится некоторая информация о разработке графических пользовательских интерфейсов для геофизического программного обеспечения. Также приводится пример того, как разрабатывался интерфейс программной системы «EMF Pro» — основные требования, визуализация данных и т.д. — Библиогр.: 1 назв.

Developing graphical user interfaces and data visualization in geophysical software / Krainikovsky S.S. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 144–149.

This article contains some information about developing graphical user interfaces in geophysical software. Also there is an example how geophysical software "EMF Pro" GUI was developed — main requirements, solutions, data visualization, etc. — Refs: 1 titles.

УДК 519.68 + 681.3.06

Использование неспецифических онтологий для хранения фактографических данных / Марчук П. А. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 150–162.

В статье описывается работа, которая велась в рамках проекта «Электронный фотоархив Сибирского отделения Российской академии наук». В проекте требовалось создать информационную систему для хранения документов и добавления к ним метаданных. — Библиогр.: 9 назв.

Using non-specific ontologies in factographic data storage / Marchuk P.A. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 150–162.

In the article, the work is described which was carried out within the framework of the "Electronic photo archive of the Siberian Branch of the Russian Academy of Sciences" project. The task was to create an information system for storage of documents and supplying them with metadata. — Refs: 9 titles.

УДК 519.68 + 681.3.06

Организация историко-культурного пространства в Интернете с использованием информационных технологий / Несговорова Г.П. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 163–173.

Обсуждаются вопросы применения информационно-коммуникационных технологий в сохранении российского и мирового культурного наследия. Даются определения, что такое культурное наследие, объект культурного наследия, предлагается классификация объектов культурного наследия, рассматриваются классические и информационные технологии их сохранения. Приводятся примеры российской и мировой сети культурного наследия. — Библиогр.: 3 назв.

Organization of the historical-cultural space in Internet using information technologies / Nesgovorova G.P. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 163–173.

Problems of informational and comminucational technologies and its use for conservation of Russian and global cultural heritage are discussed. The definitions of cultural heritage, object of cultural heritage are given, classification of cultural heritage objects and technologies of their conservation are suggested. Examples of Russian and global network of cultural heritage are presented. — Refs: 3 titles.

УДК 519.68 + 681.3.06

Внутренние представления среднего уровня для компиляторов языка Sisal / Пыжов К. А. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 174–185.

Рассматриваются технологии оптимизации использования памяти в компиляторах языка Sisal. Описываются внутренние представления среднего уровня IR2 и IR3, разработанные для компилятора Sisal 3.1 системы функционального программирования SFP, разрабатываемой в Институте систем информатики имени А.П. Ершова СО РАН. Кратко описаны механизмы построения и трансляции этих представлений, а также их оптимизирующие преобразования. — Библиогр.: 8 назв.

The middle-level internal representations for the Sisal language compilers / Pyjov K.A. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 174–185.

The paper considers some technologies of memory optimization in the Sisal language compilers. It also presents the middle-level internal representations IR1 and IR2 developed for the Sisal 3.1 compiler. This compiler is a part of SFP functional programming system which is currently being developed in the A.P. Ershov Institute of Informatics Systems SB RAS. The methods of building, translation and optimization of these representations are briefly described. — Refs: 8 titles.

УДК 519.68 + 681.3.06

Автоматная модель визуального описания синтаксического разбора / Стасенко А.П. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 186–209.

Вводится и исследуется модель автомата, подходящая для наглядного описания эффективного нисходящего синтаксического разбора языков программирования. Доказывается, что в детерминированном случае введённая автоматная модель допускает класс LL_1-языков. Разбор более широких классов языков описывается неявно с помощью контекстных состояний. Кроме того, модель включает средства иерархической обработки неопределённостей

для обработки ошибок трансляции без накладных расходов полного определения автомата. Показаны способы повышения эффективности автомата введённой модели, такие как минимизация состояний, устранение мнимых переходов и недостижимых состояний.

Automaton model of visual description of syntax parsing / Stasenko A.P. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 186–209.

The paper introduces and investigates the automaton model suitable for clear visual description of effective descending syntax analysis of programming languages. It is shown that in the deterministic case the presented automaton model allows class of LL_1 languages. The presented automaton model indirectly allows more powerful languages via its context states and supports hierarchical processing of indeterminacy for implementing syntax error handling without overhead of completely specified automaton. The paper shows the ways to improve efficiency of automaton of the presented model such as state minimization, removal of ϵ-transitions and unreachable states.

УДК 519.68 + 681.3.06

Технология автоматизации мониторинга и контроля легальности финансовых операций современных кредитных организаций / Филябин С. В. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 210–219.

В статье рассматриваются проблемы современных коммерческих банков, связанные с существенным ростом числа финансовых сервисов и контролем легальности операций. Дается описание автоматизированной системы анализа финансовых документов: архитектура и логика работы. Описаны внутренние лингвистические алгоритмы поиска и процесс интеграции с информационной системой банка. Разработка ведется в соответствии с положениями центрального банка РФ (207-П) и требованиями бизнеса. — Библиогр.: 7 назв.

Technology of automation of monitoring and control of legality of financial operations of the modern credit organizations / Filyabin S.V. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 210–219.

The article considers problems of modern commercial banks connected with an essential growth of number of financial services and control of legality of operations. The description of the automated system of analysis of financial documents is given: the architecture and logic of work. Internal linguistic algorithms of search and process of integration with information system of bank are described. Development is conducted according to regulations of the central bank of the Russian Federation (207-P) and requirements of business. — Refs: 7 titles.

УДК 519.68 + 681.3.06

О некоторых методах моделирования аппаратуры электромагнитного каротажа / Шпак М.В. // Методы и инструменты конструирования программ. — Новосибирск, 2007. — С. 220–225.

В статье описаны математические постановки задач электрического и электромагнитного каротажа, и приведен перечень методов их решения. — Библиогр.: 4 назв.

Some methods of modelling the equipment of electromagnetic well logging (carotage) / Shpak M.V. // Methods and tools of program construction. — Novosibirsk, 2007. — P. 220–225.

In the article, mathematical formulations of problems of electric and electromagnetic well logging are described and the list of methods of their solutions is given. — Refs.: 4 titles.

МЕТОДЫ И ИНСТРУМЕНТЫ КОНСТРУИРОВАНИЯ ПРОГРАММ

Под редакцией
проф. Виктора Николаевича Касьянова

Рукопись поступила в редакцию 15. 02. 2007
Ответственный за выпуск Г. П. Несговорова
Редактор Т. М. Бульонкова

Подписано в печать 27. 12. 2007
Формат бумаги 60 × 84 1/16 Объем 13,4 уч.-изд.л., 14,7 п.л.
Тираж 75 экз.

Центр оперативной печати "Оригинал 2",
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214-45-35