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

В. И. Шелехов

ВВЕДЕНИЕ В ПРЕДИКАТНОЕ ПРОГРАММИРОВАНИЕ

Препринт
100

Новосибирск 2002

Для класса программ, спецификация которых представима в виде математического предиката, предлагается формализация понятия программы в виде исчисления вычислимых предикатов. На базе исчисления строится язык предикатного программирования, продолжающий линию языков функционального программирования.

Предикатная программа может быть преобразована в эффективную императивную применением последовательности трансформаций. Обычный цикл императивной программы есть результат трансформации хвостовой рекурсии в предикатной программе. Чтобы преобразовать рекурсию к хвостовому виду, применяется универсальный метод обобщения исходной задачи. Трансформация рекурсии в цикл открывает возможность применения серии трансформаций: подстановку определения предиката на место вызова, склеивание переменных и др.

Гиперфункция оказывается адекватной формой спецификации для многих программ. Оператор расщепления, конструируемый на базе гиперфункции, определяет гибкие формы записи алгоритмов большой выразительной силы. В частности, с помощью расщеплений естественным образом реализуется обработка исключений. Как следствие, появляются процедуры с несколькими равноправными выходами. Гиперфункции и расщеплению нет аналогов ни в одном языке программирования.

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

V. I. Shelekhov

INTRODUCTION TO PREDICATE PROGRAMMING

Preprint
100

Novosibirsk 2002

Calculus of computing predicates is proposed for the programs whose specification may be represented by a predicate. A language of predicate programming is developed on the basis of the calculus.

A predicate program may be transformed into effective imperative one by applying a sequence of transformations. A cycle in an imperative program is the result of the tail recursion transformation. To transform a recursion to the tail recursion form, a universal method of generalization of a predicate program is used. After of the tail recursion transformation, other transformations may be applied: inline transformation for a predicate call, replacing two or more variables by one variable, etc.

A hyperfunction appears to be an adequate form of specification of some programs. The split statement based on a hyperfunction is proposed as a new programming construct.

ВВЕДЕНИЕ

Понятие **программы** является базисным в современной информатике. Наиболее существенными понятиями, связанными с понятием программы, являются **спецификация** и **автоматическая вычислимость** программы. Любая универсальная концепция понятия программы предполагает некоторую формализацию понятий спецификации и вычислимости программы. В анализе этих понятий наиболее общими являются вопросы о формах спецификации программы, а также вычислимых формах, которые может принимать программа. С указанными двумя вопросами связан третий — вопрос классификации **типов данных**.

В некоторых теоретических моделях спецификация программы определяется в виде математической функции, отображающей значения, поступающие на вход программы, в значения, вырабатываемые как результат исполнения программы. Однако существует много программ, спецификация которых принципиально невыразима в виде функции, например, если спецификация описывает взаимодействие совокупности параллельных процессов. Вопрос общей формы спецификации программы является открытой проблемой.

Наиболее распространенными вычислимыми формами, используемыми при построении программ, являются: суперпозиция операторов “A; B” и условный оператор “**if** C **then** A **else** B” для операторов A и B и условия C. Полный набор вычислимых форм определяется в зависимости от парадигмы программирования.

В данной работе ограничимся рассмотрением программ, спецификация которых имеет вид математического **предиката**. Спецификация определяет условие некоторой математической **задачи**. Написанию программы предшествует математическое **решение** задачи. Это решение в принципе может быть эксплицировано в виде формул на языке исчисления предикатов. Утверждается, что вычислимость программы обеспечивается благодаря вычислимости решения, а точнее, вычислимости логических формул, являющихся эксплицированной формой решения задачи. Следовательно, сначала следует исследовать свойство вычислимости логических формул.

Концепция (автоматической) вычислимости логических формул представлена в данной работе в виде **исчисления вычислимых предикатов**. Вычислимый предикат конструируется с помощью базисных вычислимых форм, называемых **базисными вычислимыми логическими компози-**

циями. Возможны следующие виды базисных композиций: суперпозиция, альтерация, параллельная композиция, применение предиката, порождение предиката, конструктор массива и расщепление.

На базе языка исчисления вычислимых предикатов строится **язык предикатного программирования**, приближенный к традиционному языку математики и более удобный для записи решений. Язык имеет имя **P** (**P**redicate programming language). Для записи вычислимых композиций используются привычные в программировании обозначения. Например, для суперпозиции используется запись: A(x,z); B(z,y), а для альтерации — **if** C(x) **then** A(x,y) **else** B(x,y) **end**.

Дальнейшей задачей является определение форм связи предикатных программ с обычными императивными программами. По нашей гипотезе, для всякой разумно написанной императивной программы, спецификация которой имеет форму предиката, существует решение на языке P, из которого она может быть получена применением последовательности **трансформаций** предикатной программы с последующим оформлением полученной программы в требуемом языке программирования.

Установлено, что циклы, за исключением параллельных, получаются как результат преобразования **хвостовой** рекурсии (tail-рекурсии в языке Лисп [10]) в определениях предикатов. Рекурсия в наиболее простом решении задачи обычно не является хвостовой. Существует ряд подходов по приведению рекурсии к хвостовому виду, как правило, автоматическими преобразованиями. В настоящей работе предлагается универсальный метод **обобщения** исходной задачи для получения решения с хвостовой формой рекурсии. Этот метод является ключевым для эффективной реализации предикатных программ.

Замена хвостовой рекурсии циклом является стартовой трансформацией в процессе преобразования предикатной программы в императивную. Такая замена открывает возможность применения ряда трансформаций: подстановку одного определения в другое, склеивание переменных, заменяющее несколько разных переменных одной, и других. В полученной программе циклы, определяемые возвратными операторами перехода, могут быть оформлены операторами **loop**, **while** и **for**. В итоге получаем вполне эффективную программу, которая могла бы быть написана на языке типа Паскаль.

Особенностью императивных программ является использование указателей при кодировании сложных структурных данных. Язык P предоставляет средства рекурсивного определения структурных типов данных, что обеспечивает запись решения без использования указателей.

В настоящей работе определяется базис предикатного программирования, исследуются методы программирования и трансформации предикатных программ для получения эффективных императивных программ. Материал иллюстрируется на примерах задач.

В разд. 1 рассматриваются понятия спецификации и автоматической вычислимости программы. Анализируется связь между вычислимостью программы и вычислимостью логических формул. Определяется исчисление вычислимых предикатов.

В разд. 2 излагается базис предикатного программирования. Определяются основные виды базисных вычислимых композиций: суперпозиция, альтерация, параллельная композиция, применение предиката, порождение предиката и конструктор массива; исследуются их свойства. На базе этих видов композиций последовательно определяются конструкции языка Р: операторы, выражения, типы и массивы.

В разд. 3 определяется метод обобщения исходной задачи для преобразования рекурсии к хвостовому виду. Определяется базисный набор трансформаций предикатной программы для получения эффективной императивной. В разделе 4 определяются производные формы конструктора массива.

В разд. 5 определяется **гиперфункция** как новый вид спецификации, выразимой в виде предиката. Сначала исследуется типовая ситуация на примере задачи решения системы линейных уравнений методом Гаусса. На гиперфункции базируется новый вид базисных вычислимых логических композиций: **расщепление**, обеспечивающее свободу и гибкость построения предикатных программ, которые недоступны даже в развитых языках императивного программирования.

В разд. 6 на базе гиперфункций и расщеплений определяются структурные типы данных, в частности, кортежи, объединения и последовательности. В разд. 7 новые возможности иллюстрируются на примере одной задачи обработки последовательности.

В разд. 8 иллюстрируются особенности предикатного программирования на примере известной задачи реализации быстрого преобразования Фурье.

Обзор работ, связанных с проблематикой предикатного программирования, представлен в разд. 9. В заключении суммируются положения данной работы и рассматриваются перспективы дальнейшего развития.

1. ИСЧИСЛЕНИЕ ВЫЧИСЛИМЫХ ПРЕДИКАТОВ

Понятие **программы** является базисным в современной информатике. Начнем анализ понятия программы с фиксации двух его определяющих свойств.

Вычисление по программе применяется для преобразования информации в цепочке действий некоторого материального процесса. Преобразование информации реализуется в соответствии с назначением (целью) процесса. Описание преобразования информации, реализуемого вычислением программы, называется **спецификацией** программы. Спецификация описывает связь между информацией, поступающей на вход программы, и информацией, получаемой как результат исполнения программы. Связь, описываемая спецификацией, имеет объективный характер и отражает реально существующие отношения между объектами процесса. Будем использовать расширительную трактовку, понимая под спецификацией также и ту связь, которую она описывает. **Первое свойство** заключается в следующем: всякая программа имеет спецификацию, причем спецификация первична по отношению к программе. Отметим, что спецификация как описание может в действительности отсутствовать, быть неполной или ошибочной.

Второе свойство — **автоматическая вычислимость** программы. Программа представляется текстом на некотором формальном языке программирования. Язык определяет набор языковых конструкций, правила построения программы из конструкций и правила исполнения всех видов конструкций и программы в целом. Семантика языка подразумевает наличие процессора, исполняющего программу. Как следствие, программа может быть оттранслирована на язык одной из существующих ЭВМ и автоматически исполнена на ней.

Программы делятся на два класса: вычислительные и системные. **Вычислительная программа** реализует решение некоторой математической задачи; спецификация программы определяется условиями задачи и в принципе может быть эксплицирована в виде математического **предиката**. Спецификация **системной программы** является результатом определения требований к разработке системной программы. Как правило, ее спецификация описывает взаимодействие совокупности параллельных процессов, что нельзя выразить в виде предиката. Противопоставление вычислительных и системных программ относительно. Вычислительная программа может включать системные части. Спецификации частей системной программы могут быть представлены в виде предиката.

В данной работе ограничимся рассмотрением программ, спецификация которых имеет вид математического **предиката**. Системные программы существенно сложнее вычислительных, и их исследование целесообразно провести позднее, используя базис, полученный при изучении вычислительных программ.

Предметом дальнейшего анализа понятия программы будет детализация свойства автоматической вычислимости. С этой целью раскроем связь между спецификацией и программой.

Связь спецификации и программы опосредована через **решение** математической **задачи.** Задача определяет исходные величины (**входные** значения программы) и неизвестные (**результирующие** значения программы). Условие задачи есть спецификация, связывающая исходные и неизвестные величины. **Решение** определяет эффективную процедуру вычисления неизвестных величин. Кроме того, решение представляет совокупность утверждений (следствий), на базе которых строится процедура вычисления. Решение реализует также вывод (доказательство) утверждений из спецификации. Программа есть формальная запись процедуры вычисления на языке программирования. Исполнение программы реализует автоматическое вычисление результирующих значений, удовлетворяющих спецификации, по входным значениям.

Решение, даже записанное на строгом математическом языке, не может быть использовано для автоматического вычисления, т.е. не является программой. Наоборот, программа, хотя и является носителем логических связей реализуемого решения, не может быть объектом логических (математических) манипуляций, обычно проводимых в процессе решения задачи, т.е. не может быть использована в качестве решения. Тем не менее, при разработке программы и ее модификации программист анализирует текст программы для реализации следующих действий:

- проверка того, соответствует ли программа (или ее часть) решению;
- восстановление спецификации программы или ее части при неполноте описания спецификации или ее отсутствии.

Так или иначе, программист вынужден оперировать логическими свойствами программы, но не в математическом стиле. Как следствие этого, ошибки в программировании, для устранения которых применяются отладка и тестирование, что, однако, не гарантирует избавления от ошибок. Отметим, что многочисленные попытки автоматизировать логические действия с программами оказались безуспешными.

Решение, в принципе, может быть эксплицировано в виде формул на языке исчисления предикатов. Утверждается, что вычислимость программы обеспечивается благодаря потенциальной вычислимости решения, а точнее, вычислимости логических формул, являющихся эксплицированной формой решения задачи. Следовательно, сначала следует исследовать свойство автоматической вычислимости логических формул.

В качестве примера рассмотрим определение суперпозиции двух функций, являющееся аналогом суперпозиции операторов “A; B” в программировании:

$$C(x,y) \equiv (\exists z)(A(x,z)\&B(z,y)) \qquad (1.1)$$

В каждом из трех предикатов C, A и B первый аргумент рассматривается в качестве исходной переменной, а второй аргумент — в качестве неизвестной. Допустим, A и B являются вычислимыми предикатами: для каждого предиката по значению первого аргумента можно вычислить значение второго. Очевидно, что формула (1.1) является вычислимой. Процедура вычисления может быть сформулирована следующим образом: “Вычисляется предикат A для переменной x в качестве первого аргумента. Полученное значение второго аргумента используется в качестве первого аргумента при вычислении предиката B для получения искомого значения переменной y”.

Проанализируем, чем обусловлено свойство автоматической вычислимости формулы (1.1). Это свойство не зависит от самих предикатов A и B (достаточно лишь их вычислимости), однако зависит от связей по аргументам предикатов A и B, а также от операции конъюнкции “&”. Таким образом, свойство вычислимости (1.1) есть свойство формы логической формулы (1.1).

Вхождения произвольных вычислимых предикатов A и B в формуле (1.1) будем называть **метапараметрами** формулы. Логическая формула, в которой вхождения предикатов рассматриваются как метапараметры, называется **логической композицией**.

Рассмотрим множество **вычислимых логических композиций**, автоматически вычислимых благодаря своей форме. Если в вычислимую логическую композицию вместо вхождения метапараметра подставим другую вычислимую логическую композицию, то получим логическую формулу, которая также будет вычислима[1]. Поэтому нас будут интересовать **базис-**

[1] Доказательство этого очевидного утверждения не реализуется в общем виде, а может быть проведено лишь для конкретных видов вычислимых композиций.

ные вычислимые логические композиции, не являющиеся результатом подстановки одной вычислимой композиции в другую. Таковой является композиция (1.1), называемая **суперпозицией**. В данной работе определяются другие **виды** базисных композиций**:** альтерация, параллельная композиция, применение предиката, порождение предиката, конструктор массива и расщепление. Данная классификация вероятно неполна.

Определим **исчисление вычислимых предикатов**. Ядром исчисления являются **примитивные** вычислимые предикаты для стандартных математических операций. Любой другой предикат исчисления имеет определение вида:

⟨имя предиката⟩(⟨список переменных⟩) ≡
⟨базисная вычислимая логическая композиция⟩ ,

где в правой части на место метапараметров подставлены либо примитивные вычислимые предикаты, либо предикаты, принадлежащие исчислению, а в качестве параметров предикатов используются простые переменные.

Система определений предикатов обычно является рекурсивной. Необходимым требованием является правильное построение рекурсии, так чтобы процесс вычисления по системе был бы конечным. **Программой** в исчислении является полный набор определений для предиката, представляющего спецификацию задачи.

Постулируем следующий тезис. На языке исчисления вычислимых предикатов можно оформить любое решение любой задачи, спецификация которой определяется в виде предиката.

Язык исчисления вычислимых предикатов неудобен для представления решений. На его базе постороим **язык предикатного программирования**, приближенный к традиционному языку математики и более удобный для записи решений. Язык имеет имя **P** (**P**redicate programming language). В языке P используются:

- функциональная форма записи наряду с предикатной;
- запись в виде математических выражений;
- подстановка одних вычислимых логических композиций в другие;
- специальные обозначения для базисных вычислимых логических композиций.

Для суперпозиции используется обозначение: A(x,z); B(z,y), а для альтерации — **if** C(x) **then** A(x,y) **else** B(x,y) **end**.

Программа на языке P является полноценной математической конструкцией. Ее нетрудно преобразовать в математическую форму путем сис-

тематической замены обозначений базисных композиций на их логические прототипы.

2. БАЗИС ЯЗЫКА ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

В данном разделе определяются шесть основных видов базисных вычислимых композиций: суперпозиция, альтерация, параллельная композиция, применение предиката, порождение предиката и конструктор массива. Определяемые здесь конструкции языка **P** (**P**redicate programming language) получены на базе этих видов композиций. Представлены правила построения предикатных программ, называемых **P-программами**.

Для описания синтаксиса предикатной программы используется расширенный язык Бэкусовских нормальных форм (БНФ). Фрагмент синтаксического определения — последовательность терминальных и нетерминальных символов — может быть обрамлена квадратными или фигурными скобками:

[⟨фрагмент⟩] — означает возможное отсутствие ⟨фрагмента⟩;
{⟨фрагмент⟩} — допускает многократное повторение ⟨фрагмента⟩.

Терминальные символы {, }, [,] и ‖ будем изображать с подчеркиванием: $\underline{\{}$, $\underline{\}}$, $\underline{[}$, $\underline{]}$ и $\underline{\|}$.

Предикатная программа содержит набор вычислимых **определений предикатов**, среди которых могут находиться описания типов:

⟨предикатная программа⟩ ::=
{ ⟨описание типа⟩;| ⟨описание переменных⟩;| ⟨определение предиката⟩ }

Определение предиката имеет следующую форму:

⟨определение предиката⟩::=
⟨обозначение предиката⟩ ≡ ⟨вычислимая логическая композиция⟩

Правая часть определения является логической формулой, в которой используются другие предикаты; их вхождения называются **вызовами** предикатов. Вхождение предиката в левой части определения называется **определяющим**.

2.1. Обозначения предикатов

Предикат есть математическое условие, связывающее значения исходных и неизвестных переменных задачи. Это условие в принципе эксплицируемо в виде формулы на языке исчисления предикатов.

Допустим, предикат A(x,y) является спецификацией задачи, где A обозначает имя предиката, а x и y — произвольные наборы переменных, причем x — набор исходных переменных, а y — неизвестных. Будем использовать запись A(x: y), где ":" разделяет исходные и неизвестные (результирующие) переменные. В случае, когда задача предписывает вычислить логическое значение предиката A (набор y пуст), будем использовать в качестве результата дополнительный параметр логического типа, например b, и записывать предикат в виде A(x: b).

Для примитивных предикатов используются стандарные имена. Например, для предиката равенства c = d используется обозначение =(c: d), когда нужно вычислить значение переменной d по значению c; обозначение =(c,d: b) используется, когда требуется вычислить логическое значение b предиката c = d; операция "плюс" представляется в виде +(c,d: e); "унарный минус" представляется в виде -(c: d) и т.д.

Для вызова предиката B(x: y) допускается как предикатная, так и функциональная форма записи. B(x) обозначает значение результирующего набора "y" для исходного набора "x", т.е. B(x: y) эквивалентно y = B(x). Конструкцию B(x) будем называть **вызовом функции**. Для предиката B(x: b), где b — логическая переменная, используется функиональная форма B(x) в позиции, где требуется логическое значение предиката.

Для примитивных предикатов используются также общепринятые инфиксная и префиксная формы записи. Предикат =(c,d: b) записывается в виде c = d, предикат +(c,d: e) — в виде e = c+d, предикат -(c: d) записывается в виде d = -c.

Предикат =(c: d) условимся записывать в виде d = c, причем результирующая переменная ставится первым операндом.

В дальнейшем будем использовать два способа описания параметров при изображении определяющего вхождения предиката:

⟨обозначение предиката⟩::=

⟨имя предиката⟩ (⟨описание исходных и результирующих параметров⟩)

⟨описание исходных и результирующих параметров⟩::=

[⟨описания или обозначения параметров⟩] **:**

⟨описания или обозначения параметров⟩

За “**:**” следует описание параметров-результатов.

⟨описания или обозначения параметров⟩::=
⟨описания параметров⟩ | ⟨обозначения параметров⟩
⟨описания параметров⟩::=
⟨изображение типа⟩⟨пробел⟩⟨имя параметра⟩ [{,⟨имя параметра⟩}]
[,⟨описания параметров⟩]
⟨обозначения параметров⟩::=
⟨имя параметра⟩ [,⟨обозначения параметров⟩]
⟨имя параметра⟩::=⟨идентификатор⟩

⟨изображение типа⟩ декларирует тип одного или нескольких следующих за ним параметров (см. п. 2.5); ⟨пробел⟩ есть терминальный символ, разделяющий описание типа и имена параметров. Тип определяет множество значений, допустимых для каждого параметра.

2.2. Базисные вычислимые логические композиции

Правая часть определения предиката строится из **базисных вычислимых логических композиций**. Определим пять видов композиций: суперпозицию, альтерацию, параллельную композицию, применение предиката и порождение предиката. Правила исполнения композиций подразумевают наличие некоторого виртуального **процессора**, работающего в абстрактной **памяти**.

Будем говорить, что предикат A(x**:** y) вычисляет значения переменных набора y по значениям переменных набора x, подразумевая, что вычисление реализуется по определению предиката, а для примитивных предикатов — непосредственно виртуальным процессором. Описание исполнения вызова предиката A(x**:** y) будет дано в п. 2.5.

Далее в формулах вызовы произвольных предикатов A, B, C и других предполагаются вычислимыми. По умолчанию, переменные x, y, z и другие обозначают произвольные непустые наборы переменных, непересекающиеся в рамках одной формулы.

Композиция вида “**суперпозиция**” представляется следующим определением:

$$D(x{:}\ y) \equiv (\exists z)(A(t{:}\ z,r)\&B(z,v{:}\ p)) \qquad (2.1)$$

Наборы x, t, r и v могут быть пустыми. Наборы t и v могут пересекаться, а их объединение совпадает с набором x. Объединение наборов r и p совпа-

дает с набором y. В языке P для суперпозиции (2.1) используется следующий способ записи:

$$D(x{:}\ y) \equiv \{\langle\text{описания переменных набора } z\rangle;\ A(t{:}\ z,r);\ B(z,v{:}\ p)\} \qquad (2.2)$$

Конструкция (2.2) называется **блоком.** Переменные набора z являются **локальными** переменными блока. При исполнении конструкции (2.2) сначала исполняются описания переменных набора z, в результате чего эти переменные заводятся в памяти. Далее предикат A(t: z,r) вычисляет значения переменных наборов z и r по значениям набора t. Затем предикат B(z,v: p) вычисляет значение набора p по значениям наборов z и v. Наконец, локальные переменные набора z удаляются из памяти, и на этом исполнение блока завершается.

Композиция вида "**альтерация**" определяется следующей формулой:

$$D(x,b{:}\ y) \equiv (b \Rightarrow A(z{:}\ y))\&(\neg b \Rightarrow B(t{:}\ y))$$

Переменная b является логической. Наборы z и t могут пересекаться и быть пустыми, а их объединение совпадает с набором x. В языке P для альтерации используется способ записи:

$$D(x,b{:}\ y) \equiv \textbf{if } b \textbf{ then } A(z{:}\ y) \textbf{ else } B(t{:}\ y) \textbf{ end} \qquad (2.3)$$

Конструкция (2.3) является **условным оператором**. При ее исполнении проверяется значение логической переменной b. Если значение b истинно, предикат A(z: y) по значениям набора z вычисляет значения переменных набора y, иначе предикат B(t: y) по значениям набора t вычисляет значения переменных набора y.

В суперпозиции (2.1) набор z не может быть пустым. Теперь рассмотрим случай, когда в формуле (2.1) набор z — пуст:

$$D(x{:}\ y) \equiv A(t{:}\ r)\&B(v{:}\ p)$$

Для наборов переменных здесь реализуются те же условия, что и для (2.1). Данная композиция уже не является суперпозицией, однако попрежнему остается вычислимой логической композицией, в которой оба конъюнкта могут вычисляться независимо. Композицию назовем "**параллельной**". В языке P будем использовать запись:

$$D(x{:}\ y) \equiv \{A(t{:}\ r)\,||\,B(v{:}\ p)\} \qquad (2.4)$$

Конструкция (2.4) называется **параллельным оператором**. Ее исполнение реализуется следующим образом. Предикат A(t: r) по значениям переменных набора t вычисляет значения набора r. Предикат B(v: p) по значениям переменных набора v вычисляет значения набора p. Оба вычисления совершаются независимо друг от друга. Вычисление конструкции (2.4) закончится, когда завершатся оба вычисления. В данном описании мы не делаем никаких предположений об устройстве виртуального процессора, обеспечивающего параллельные вычисления оператора (2.4).

Композиция вида "**применение предиката**" имеет определение:

$$D(x,A: y) \equiv A(x: y) \tag{2.5}$$

Предикат A является входным параметром определяемого предиката D; набор x может быть пустым. Исполнение данной конструкции заключается в исполнении вызова предиката A(x: y).

Композиция вида "**порождение предиката**" имеет определение:

$$D(x: A) \equiv (\forall y \in Y)(\exists z)(A(y: z) \equiv B(x,y: z))$$

Наборы x и y могут быть пустыми, параметр A — переменная типа "предикат", Y есть набор типов, соответствующий набору y. Предикат B является метапараметром формулы. В языке P будем использовать запись:

$$A = \textbf{lambda}(y: z)\ B(x,y: z) \tag{2.6}$$

Исполнение конструкции "порождение предиката" заключается в построении определения нового предиката A_0:

$$A_0(y: z) \equiv \{\langle \text{Описания переменных набора x} \rangle;\ x = x_0;\ B(x,y: z)\}$$

Набор x_0 обозначает набор значений для переменных набора x в момент вычисления определения предиката D(x: A). Результатом исполнения является присваивание переменной A предиката A_0 в качестве значения.

2.3. Свойства вычислимых композиций

Подстановка правой части определения вычислимого предиката в другую вычислимую композицию на место вызова этого предиката дает новую логическую композицию, которая является вычислимой.

Определим условия, при которых конъюнкция вычислимых предикатов A и B с произвольными наборами переменных x, y, z и t является вычислимой:

$$D(u: v) \equiv A(x: y)\&B(z: t) \tag{2.7}$$

Если наборы y и t пересекаются, то композиция (2.7) не может быть вычислимой. Поэтому далее будем считать, что наборы y и t не пересекаются. Если набор y пересекается с z, а набор t пересекается с x, то (2.7) также не является вычислимой. Если набор y не пересекается с z и набор t не пересекается с x, то (2.7) — вычислима и является параллельной композицией. Наконец, если набор y пересекается с z, а набор t не пересекается с x (или наоборот, первая пара не пересекается, а вторая — пересекается), то (2.7) вычислима и является суперпозицией.

Параллельная композиция является коммутативной и ассоциативной, поскольку таковой является конъюнкция. Поэтому далее композицию {{A(...)||B(...)}||C(...)} будем записывать без внутренних скобок в виде {A(...)||B(...)||C(...)}. Как следствие, определение параллельной композиции обобщается на произвольное число предикатов; при исполнении параллельной композиции все эти предикаты могут вычисляться независимо друг от друга.

Суперпозиция не является коммутативной. Для анализа свойства ассоциативности рассмотрим композицию {{A(...);B(...)};C(...)}. Имеет место:

$$\{\{A(...);B(...)\};C(...)\} \equiv \{\{A(...)\&B(...)\}\&C(...)\} \equiv \{A(...)\&\{B(...)\&C(...)\}\}.$$

Используя рассмотренные выше свойства вычислимости конъюнкции предикатов, можно показать, что формула {B(...)&C(...)}, а также формула {A(...)&{B(...)&C(...)}} являются вычислимыми.

Формула {A(...)&{B(...)&C(...)}} в итоге определяет одну из двух композиций: {A(...); {B(...); C(...)}} или {A(...); {B(...) ‖ C(...)}}. Отметим, что для композиции {A(...); {B(...);C(...)}} изменение расстановки внутренних скобок приводит к аналогичным результатам. Условимся, что суперпозиция “связывает” сильнее, чем параллельная композиция; т.е. вместо {{A(...); B(...)}||C(...)} будем писать {A(...); B(...) ‖ C(...)}. Будем писать {A(...); B(...); C(...)} в случае, когда суперпозиция не вырождается в параллельную композицию при любой расстановке внутренних скобок. Введенные правила обобщаются на произвольное число предикатов и позволяют записывать цепочки суперпозиций и параллельных композиций без внутренних скобок.

Если в композиции {A(x: y); {B(z: t)□C(u: v)}} набор y пересекается с набором z и не пересекается с набором u, то композиция эквивалентна {{A(x: y); B(z: t)}□C(u: v)} или, с учетом соглашения о приоритетах, композиции {A(x: y); B(z: t)□C(u: v)}. Аналогичным образом композиция {{A(x: y) □B(z: t)}; C(u: v)} эквивалентна {A(x: y)□B(z: t); C(u: v)}, если наборы y и u не пересекаются.

2.4. Производные конструкции

В суперпозиции (2.1) рассмотрим случай, когда набор r — пуст:

$$D(x{:}\ y) \equiv (\exists z)(A(t{:}\ z)\&B(z,v{:}\ y)) \qquad (2.8)$$

Наборы x, t и v могут быть пустыми. Наборы t и v могут пересекаться, а их объединение совпадает с набором x. Используя для A(t: z) функциональную форму записи z = A(t) и проведя подстановку A(t) вместо z в (2.8), получим: D(x: y) ≡ B(A(t),v: y). Далее можно написать:

$$y = D(x) = B(A(t),v) \qquad (2.9)$$

Это соответствует классическому определению суперпозиции функций.

Отметим, что при непустом наборе r формула (2.1) не может являться определением суперпозиции функций. Это лишь одна из ситуаций, где предикатное программирование оказывается шире функционального.

Применение суперпозиции (2.9) для стандартных арифметических операций, соответствующих примитивным предикатам, приводит нас к известному в математике и программировании понятию **выражения**, первичными конструкциями которого являются переменные и вызовы функций. Например,

$$Y = +(*(a,b),c) = (a*b)+c$$

Условимся далее писать произвольные выражения на месте любой входной переменной в вызове предиката. При написании выражений будем опускать избыточные скобки, применяя правила приоритетов операций. Для выражения в позиции входной переменной предиката будем использовать термин **аргумент предиката**.

Имеет место следующее тождество:

$$\{A(x{:}\ y)\square B(z{:}\ t)\};C(y,t{:}\ u) \equiv C(A(x),B(z){:}\ u)$$

Из тождества следует, что разные аргументы одного предиката могут вычисляться независимо (параллельно).

Имеет место тождество:

C(x: b); **if** b **then** A(z: y) **else** B(t: y) **end** ≡
if C(x) **then** A(z: y) **else** B(t: y) **end**

Учитывая это, для альтерации в позиции между **if** и **then** будем использовать произвольные логические выражения.

Композицию вида:

if D **then** A **else** {**if** E **then** B **else** C **end**} **end**

далее будем изображать следующим образом:

if D **then** A **elsif** E **then** B **else** C **end**

2.5. Операторы. Выражения. Типы

Сформулированные выше определения и соглашения являются базой при определении конструкций языка Р. Продолжим определения, начатые в пп. 2 и 2.1.

⟨определение предиката⟩::=⟨обозначение предиката⟩ ≡ ⟨оператор⟩
⟨оператор⟩::= ⟨предикат равенства⟩ |
⟨вызов предиката⟩ |
[↓]⟨вычислимая логическая композиция⟩ [↓]

Отличие правила ⟨определение предиката⟩ от аналогичного, приведенного в начале п. 2, обусловлено введением выражений. Условимся опускать внешние скобки "{" и "}" в изображении композиции там, где это не приводит к двусмысленности.

⟨предикат равенства⟩::=⟨переменная⟩=⟨выражение⟩

⟨переменная⟩ в предикате равенства является результатом; при исполнении переменной присваивается значение выражения.

⟨вычислимая логическая композиция⟩::=
⟨блок⟩ |
⟨условный оператор⟩ |
⟨оператор выбора⟩ |
⟨параллельный оператор⟩ |
⟨применение предиката⟩ |
⟨конструктор массива⟩ |
⟨оператор расщепления⟩

Конструктор массива определен в п. 2.6, а оператор расщепления в п. 5.3.

⟨блок⟩::=[{⟨описание переменных⟩;}]⟨оператор⟩;⟨продолжение блока⟩
⟨продолжение блока⟩::=⟨оператор⟩ | ⟨блок⟩

Блок определяет цепочку суперпозиций. Блок обычно обрамляется фигурными скобками, которые могут опускаться, если блок используется в

позициях условного или параллельного операторов. По завершению исполнения блока переменные, описанные внутри блока, удаляются из памяти.

⟨условный оператор⟩::=
if ⟨выражение⟩ **then** ⟨оператор⟩
[{ **elsif** ⟨выражение⟩ **then** ⟨оператор⟩ }]
else ⟨оператор⟩ **end**

Условный оператор может содержать произвольное число фрагментов: **elsif** ⟨выражение⟩ **then** ⟨оператор⟩, что соответствует многократной подстановке альтерации в позицию между **else** и **end.**

⟨оператор выбора⟩::=
case ⟨выражение⟩ **of**
⟨альтернативы выбора⟩
[**else** ⟨оператор⟩] **end**

Оператор выбора является другой формой условного оператора.

⟨альтернативы выбора⟩::=
⟨метка альтернативы⟩ **:** ⟨оператор⟩ {⊥ ⟨альтернативы выбора⟩}
⟨метка альтернативы⟩::=
⟨изображение константы⟩ |
⟨изображение константы⟩**..**⟨изображение константы⟩

Исполнение оператора выбора реализуется следующим образом. Вычисляется ⟨выражение⟩, и его значение последовательно сравнивается со значениями меток альтернатив. Если метка представлена диапазоном, проверяется попадание значения в этот диапазон. Далее исполняется оператор альтернативы, метка которого соответствует вычисленному значению. Если ни одна метка не соответствует значению ⟨выражения⟩, исполняется оператор после **else**. Если набор значений меток полностью покрывает тип ⟨выражения⟩, альтернатива **else** может отсутствовать**.**

⟨параллельный оператор⟩::=
⟨оператор⟩□
⟨продолжение параллельного оператора⟩
⟨продолжение параллельного оператора⟩::=
⟨оператор⟩ |
⟨параллельный оператор⟩

Параллельный оператор состоит из произвольного числа операторов, разделенных “||”, которые выполняются независимо друг от друга; выполнение параллельного оператора заканчивается по завершении исполнения всех составляющих его операторов.

⟨применение предиката⟩::=⟨вызов предиката⟩
⟨вызов предиката⟩::=
⟨имя предиката⟩ ([⟨аргументы⟩]:⟨результаты⟩) |
⟨выражение⟩ ([⟨аргументы⟩]:⟨результаты⟩)

Значением ⟨выражения⟩ в позиции вызова предиката является некоторый предикат.

⟨аргументы⟩::=⟨список выражений⟩
⟨список выражений⟩::=⟨выражение⟩ [,⟨список выражений⟩]
⟨результаты⟩::=⟨переменная⟩ [,⟨результаты⟩]

Рассмотрим вызов предиката A(e: z), где e — набор выражений, а z — набор переменных. Допустим, имеется определение предиката A(x: y) ≡ S, где S — некоторый оператор. Исполнение вызова A(e: z) эквивалентно исполнению следующего блока:

{ ⟨описания переменных для наборов x и y⟩; x = e; S; z = y },

где x = e и z = y понимаются как предикаты равенства, определенные для наборов переменных. Исполнение предиката x = e реализуется следующим образом: независимо (параллельно) вычисляются все выражения набора e — результатом такого вычисления является набор значений; далее значения полученного набора одновременно присваиваются соответствующим переменным набора x. Исполнение z = y реализуется аналогично.

Более точно, исполнение вхождения A(e: z) реализуется в виде следующей последовательности действий:

- вычисляется набор значений для набора выражений e;
- создается новая секция памяти для определения предиката A(x: y) ≡ S;
- исполняются описания переменных набора x: каждая переменная создается в секции и в момент создания ей присваивается соответствующее значение из набора, вычисленного для e;
- исполняются описания переменных набора y, переменные создаются в секции;
- исполняется оператор S;
- извлекается набор значений для переменных набора y;

– ликвидируется секция памяти вместе с переменными, содержащимися в ней;
– реализуется присваивание значений набора, извлеченного из y, соответствующим переменным набора z.

Первое и последнее действия реализуется при исполнении A(e: z), остальные действия составляют исполнение определения A(x: y) ≡ S.

Если A(e: z) является вызовом примитивного предиката, то его исполнение реализуется непосредственно виртуальным процессором. Первоначально вычисляется набор значений для e; результирующие значения присваиваются переменным набора y. Виды примитивных предикатов здесь не фиксируются.

⟨выражение⟩::= ⟨первичное⟩ |
⟨выражение⟩⟨знак бинарной операции⟩⟨выражение⟩ |
⟨знак унарной операции⟩⟨выражение⟩

Результатом исполнения выражения является значение или набор значений. Набор более чем из одного значения допустим лишь в соответствующей позиции аргумента предиката или вызова функции. При записи выражений используются правила приоритетов операций, что позволяет опускать круглые скобки. Набор операций содержит известные арифметические и логические операции, традиционно используемые в языках программирования.

⟨первичное⟩::= ⟨переменная⟩ |
⟨изображение константы⟩ |
⟨вызов функции⟩ |
[↓]⟨оператор⟩ [↓] |
⟨имя предиката⟩ |
⟨производные формы конструктора массива⟩ |
⟨порождение предиката⟩ |
(⟨выражение⟩)

Константа определяет фиксированное значение некоторого типа. Предполагается, что для любого значения, для которого существует изображение, имеется примитивный вычислимый предикат равенства: x = ⟨изображение константы⟩, а конструкция ⟨изображение константы⟩ есть функциональная форма записи этого предиката.

Значением конструкции ⟨имя предиката⟩ является обозначаемый именем предикат. Это значение может быть присвоено переменной типа “предикат”.

Результатом вычисления оператора в позиции выражения является значение результирующей переменной оператора, а если результирующих переменных несколько, то — набора значений. Как правило, оператор в позиции выражения обрамлен фигурными скобками. Значением оператора порождения предиката в позиции выражения является предикат.

⟨порождение предиката⟩::=
lambda (⟨описание исходных и результирующих параметров⟩)
[{]⟨оператор⟩ [}]

Исполнение конструкции порождения предиката реализует построение определения нового предиката. Правой частью этого определения будет ⟨оператор⟩ при фиксации в нем значений свободных переменных на момент начала исполнения конструкции порождения предиката. Новый предикат становится итоговым значением конструкции порождения предиката.

Описание производных форм конструктора массива см. в п. 4.

⟨переменная⟩::= ⟨простая переменная⟩ |
⟨элемент массива⟩ | …
⟨простая переменная⟩::=⟨имя переменной⟩

Результатом исполнения является значение переменной. Элемент массива определен в п. 2.6, другие виды переменных — в п. 6.

⟨вызов функции⟩::= ⟨имя предиката⟩ (⟨аргументы⟩) |
⟨выражение⟩ (⟨аргументы⟩)

Исполнение вызова функции аналогично исполнению вызова предиката: реализуется та же последовательность действий, описанная выше, кроме последнего действия. Полученное в итоге значение (или набор значений) является результатом вычисления вызова функции.

⟨описание переменных⟩::=
⟨изображение типа⟩⟨пробел⟩⟨имя переменной⟩ [=⟨выражение⟩] |
⟨изображение типа⟩⟨пробел⟩⟨имя переменной⟩ [{,⟨имя переменной⟩}]

⟨пробел⟩ обозначает терминальный символ “пробел”. Результатом исполнения является создание в памяти одной или нескольких переменных, имена которых указаны в описании. При наличии инициализации вычисленное значение ⟨выражения⟩ присваивается переменной. Описание переменной с инициализацией “⟨изображение типа⟩ z = ⟨выражение⟩” используется как

сокращение для “⟨изображение типа⟩ z; z = ⟨выражение⟩” в композиции суперпозиции.

⟨описание типа⟩::=

type ⟨имя типа⟩ [(⟨описания или обозначения параметров⟩)]
= ⟨изображение типа⟩

⟨имя типа⟩::=⟨идентификатор⟩

Правила для ⟨описания или обозначения параметров⟩ даны в п. 2.1. Описание типа связывает имя типа с его определением.

Тип может быть **параметризован** , т.е. зависеть от набора значений переменных, указанных в качестве параметров.

⟨изображение типа⟩::= ⟨изображение примитивного типа⟩ |
⟨изображение подмножества типа⟩ |
⟨изображение типа предиката⟩ |
⟨имя типа⟩ [(⟨аргументы⟩)] |
⟨изображение структурного типа⟩

⟨изображение примитивного типа⟩::=
nat | **int** | **real** | **bool** | **char** | **complex** |…

Описатель **nat** обозначает множество натуральных чисел, **char** — множество символов некоторого алфавита.

⟨изображение подмножества типа⟩::=
{⟨описание переменной⟩ : ⟨выражение⟩} |
⟨изображение диапазона⟩|
{⟨список частей подмножества типа⟩}

Конструкция {T x: ⟨выражение⟩} определяет подмножество типа T. Описание переменной x действует в пределах этой конструкции. Множество значений переменной x, для которых логическое ⟨выражение⟩ имеет значение “истина”, является определяемым подмножеством типа T. Если ⟨выражение⟩ содержит другие переменные, то все они являются **параметрами** изображаемого типа.

В качестве примера рассмотрим изображение типа для диапазона натуральных чисел от 1 до n+1, где n является параметром:

{**nat** i: i >= 1 & i <= n+1}

Для диапазонов будем использовать специальную синтаксическую форму:

⟨изображение диапазона⟩::=⟨выражение⟩..⟨выражение⟩

Изображение типа для приведенного примера может быть дано в более компактном виде: 1..n+1.

⟨список частей подмножества типа⟩::=
⟨выражение⟩ [,⟨список частей подмножества типа⟩] |
⟨изображение диапазона⟩ [,⟨список частей подмножества типа⟩]

Вычисленное значение ⟨выражения⟩ определяет одно значение, принадлежащее подмножеству типа. Определяемое подмножество объединяет значения всех частей списка.

Если изображение подмножества типа используется в левой части описания типа, то параметры подмножества типа должны быть указаны как параметры описываемого типа. Например:

```
type Diapason(int n) = 1..n+1;
```

Параметризованный тип может быть использован для определения другого типа непосредственно в виде ⟨имя типа⟩ (⟨аргументы⟩) или в составе структурного типа. Если в ⟨аргументах⟩ используются переменные, то эти переменные должны быть параметрами определяемого типа.

⟨изображение типа предиката⟩::=
predicate ([⟨изображения типа⟩]:⟨изображения типа⟩)
⟨изображения типа⟩::=⟨изображение типа⟩[,⟨изображения типа⟩]

Изображение типа предиката используется для описания переменной предикатного типа.

⟨изображение структурного типа⟩::=⟨изображение типа массива⟩ |…

Структурный тип определяется в виде композиции других типов, называемых **компонентными** по отношению к структурному типу. Кроме массивов имеются кортежи, объединения и другие виды структурных типов. Массивы определены в п. 2.6, остальные — в п. 6.

Структурный тип определяется двумя **фундаментальными предикатами: Comp** и **Cons**. Предикат **Comp** связывает произвольное значение структурного типа с соответствующими значениями компонентных типов. Предикат **Cons**, называемый **конструктором**, по набору значений компонентных типов строит соответствующее значение структурного типа.

2.6. Массивы

⟨изображение типа массива⟩::=**array** ⟨тип индексов⟩ **of** ⟨тип элементов⟩
⟨тип индексов⟩::=⟨изображение типа⟩
⟨тип элементов⟩::=⟨изображение типа⟩

Значение массива состоит из совокупности элементов. Каждый элемент доступен по индексу в массиве. Тип индексов должен быть конечным.

Допустим, имеется описание типа массива AR с элементами типа T и индексами типа I: **type** AR = **array** I **of** T. Фундаментальный предикат Comp для массива A и индекса i определяет элемент e массива A по индексу i: Comp(A,i: e). Comp является примитивным предикатом. Запись A[i] является обозначением для Comp(A,i) и называется **элементом массива**.

В P-программе в качестве переменной может быть использована конструкция:

⟨элемент массива⟩::=⟨переменная⟩[⟨выражение⟩]

Значением конструкции ⟨переменная⟩ является массив, ⟨выражение⟩ определяет значение индекса, а предикат Comp реализует вычисление значения соответствующего элемента массива.

Конструктор массива есть предикат Cons(x: A), реализующий создание объекта (значения типа массив) посредством создания всех элементов массива; где x — набор переменных, A — переменная типа **array** I **of** T. Произвольный i-й элемент массива вычисляется через предикат G(i,x: t), в котором i — индекс типа I и t — i-й элемент массива типа T. Конструктор Cons(x: A) определяется как базисная вычислимая логическая композиция, называемая **конструктор массива:**

$$\mathrm{Cons}(x\colon A) \equiv (\forall i \in I)\mathrm{G}(i,x\colon A_i)\ , \tag{2.10}$$

где A_i обозначает i-й элемент массива, предикат G является метапараметром формулы. Далее для A_i будем использовать обозначение A[i], понимая при этом, что такое обозначение имеет иной смысл по сравнению с введенным выше для Comp(A, i).

Композиция (2.10) имеет следующий способ записи в языке P:

forAll i∈I **do** G(i,x: A[i]) **end** (2.11)

Исполнение конструктора массивов (2.11) реализуется следующим образом. В памяти создается массив как значение типа **array** I **of** T. Для типа I

реализуется итерация элементов множества I. Для каждого значения индекса i вычисляется предикат G(i,x: A_i), где A_i обозначает i-ый элемент созданного массива. Вычисление предиката G(i,x: A_i) для разных индексов i реализуется независимо, т.е. параллельно. Когда вычисление G(i,x: A_i) будет завершено для всех индексов, созданный массив присваивается переменной A.

Будем также использовать следующую форму записи:

forAll i∈I **do** A[i]=G(i,x) **end**

В случае, когда тип I является диапазоном m..n, будем использовать другую форму записи:

forAll i=m..n **do** G(i,x: A[i]) **end** (2.12)

Наконец, определим правила для оператора "конструктор массива":

⟨конструктор массива⟩::=
forAll ⟨итератор⟩ **do** ⟨оператор⟩ **end** |
forAll ⟨итератор⟩ **do** ⟨элемент массива⟩=⟨выражение⟩ **end**
⟨итератор⟩::= ⟨переменная⟩∈⟨изображение типа⟩ |
⟨переменная⟩=⟨выражение⟩..⟨выражение⟩

Описание производных форм конструктора массива см. в п. 4.

В дальнейшем в качестве типа индексов I может быть использован кортеж двух диапазонов. Элемент массива A[(i,j)] будем записывать без круглых скобок: A[i,j].

2.7. Предикатная программа

Программа на языке P есть полная система вычислимых определений предикатов для предиката, представляющего спецификацию задачи. В полной системе определений для любого использующего вхождения имени непримитивного предиката существует его определение. Среди определений предикатов могут встречаться описания типов и переменных (см. начало п. 2).

Исполнение предикатной программы — это исполнение некоторого не принадлежащего программе вызова предиката с аргументами, задаваемыми внешним по отношению к программе образом. Исполнение вызова предиката реализует исполнение определения этого предиката. Исполнение программы завершается по окончании исполнения определения предиката и

присваивания итоговых значений результирующим переменным вызова предиката.

Система определений предикатной программы обычно является рекурсивной. При исполнении программы реализуется рекурсивное исполнение определений предикатов. Необходимым требованием является правильное построение рекурсии в системе определений, так чтобы процесс исполнения был бы конечным.

Переменные, описания которых встречаются перед определениями предикатов, называются **глобальными**. В нижеследующих определениях предикатов эти переменные являются входными параметрами, которые для сокращения записи не упоминаются при описании параметров.

Подмножество языка Р без выражений, без возможности подстановки одной вычислимой композиции в другую и без использования функционального стиля обозначений соответствует **исчислению вычислимых предикатов**. Иначе говоря, в правой части определения предиката разрешается одна из базисных вычислимых композиций, а в качестве аргументов допускаются только переменные. Любая программа на языке Р может быть автоматически преобразована в соответствующую программу в этом исчислении.

3. МЕТОД ОБОБЩЕНИЯ ИСХОДНОЙ ЗАДАЧИ. СИСТЕМА ТРАНСФОРМАЦИЙ ПРЕДИКАТНОЙ ПРОГРАММЫ

Исчисление вычислимых предикатов и базирующийся на нем язык предикатного программирования представлены выше как определение понятия автоматической вычислимости программ, спецификация которых имеет форму предиката. Дальнейшей задачей исследования понятия программы является анализ конкретных форм связи предикатных программ с обычными императивными программами. По нашей гипотезе, для всякой разумно написанной императивной программы, спецификация которой имеет форму предиката, существует решение на языке Р, из которого программа может быть получена применением последовательности **трансформаций** Р-программы с последующим оформлением полученной программы в требуемом языке программирования. Для проведения трансформаций предикатных программ будет определено императивное расширение языка Р операторами присваивания, перехода, циклами **loop**, **while** и **for** и другими конструкциями.

Установлено, что циклы за исключением параллельных получаются как результат трансформации **хвостовой** рекурсии [10] в определениях предикатов. Рекурсия в наиболее простом решении задачи обычно не является хвостовой. Существует ряд подходов по приведению рекурсии к хвостовому виду, как правило, автоматическими преобразованиями. В настоящей работе предлагается универсальный метод **обобщения** исходной задачи для получения решения с хвостовой формой рекурсии.

Метод обобщения задачи является ключевым для эффективной реализации предикатных программ.

Замена хвостовой рекурсии циклом является стартовой трансформацией в процессе преобразования Р-программы в императивную программу. Такая замена открывает возможность применения ряда преобразований: подстановку одного определения в другое, склеивание переменных, заменяющее несколько разных переменных одной, и других. В полученной программе циклы, определяемые возвратными операторами перехода, могут быть оформлены операторами **loop**, **while** и **for**. В итоге получаем вполне эффективную программу, которая могла бы быть написана на языке типа Паскаль или С.

В данном разделе описывается метод обобщения исходной задачи, императивное расширение языка Р и набор базовых трансформаций. Изложение иллюстрируется на примерах простых задач.

3.1. Групповой оператор присваивания

Очевидно, что оператор присваивания возникает из предиката равенства, однако не только из него. При описании правил исполнения вызова предиката (см. п. 2.5) был введен предикат равенства более общего вида x = e, где x — набор переменных, а e — набор выражений. Его аналогом в императивном программировании является **групповой оператор присваивания**, возникающий при трансформации предикатной программы.

⟨групповой оператор присваивания⟩::=
⌊⟨список переменных⟩⌋ := ⌊⟨список выражений⟩⌋
⟨список переменных⟩::=⟨переменная⟩ [,⟨список переменных⟩]
⟨список выражений⟩::=⟨выражение⟩ [,⟨список выражений⟩]

Исполнение реализуется следующим образом. Определяется (вычисляется) список переменных в левой части. Независимо (параллельно) вычисляется каждое из выражений в списке правой части. Значения из получен-

ного набора одновременно присваиваются соответствующим переменным в левой части.

Часто групповой оператор присваивания необходимо заменить набором обычных операторов присваивания. Это преобразование называется **раскрытием** группового оператора присваивания. В общем случае раскрытие возможно при использовании дополнительных промежуточных переменных. Например, раскрытие оператора |a,b|:=|b,a| реализуется через дополнительные переменные t1 и t2. Поскольку |a,b|:=|b,a| эквивалентно "|t1,t2|:=|b,a|; |a,b|:=|t1,t2|", последнее можно заменить цепочкой обычных присваиваний:

t1:=b; t2:=a; a:=t1; b:=t2;

Для раскрытия |a,b|:=|b,a| достаточно использования одной дополнительной переменной:

t2:=a; a:=b; b:=t2;

В большинстве случаев раскрытие возможно без использования дополнительных переменных, однако для этого часто требуется поменять местами операторы присваивания. Например, раскрытие |a,b|:=|c,a| реализуется присваиваниями: b:=a; a:=c.

3.2. Замена хвостовой рекурсии циклом

Предикатная программа будет существенно проигрывать в эффективности по сравнению с аналогичной императивной программой. Одной из причин этого является неэффективность исполнения вызова предиката из-за подстановки "значением" аргументов и результатов структурных типов. Подстановка определения предиката на место вызова повышает эффективность предикатной программы. Однако если определение предиката является рекурсивным, подобная подстановка обычно не приносит пользы.

Существует специальный случай рекурсии, называемый **хвостовой рекурсией (**tail-рекурсией в языке Лисп [10]), когда можно заменить рекурсию циклом. Вызов предиката определяет хвостовую рекурсию, если:

- имя вызываемого предиката совпадает с именем определяемого предиката, в котором находится вызов;
- вызов является последней исполняемой конструкцией в определении.

Второе условие требует уточнения.

В качестве примера рассмотрим задачу D(a,b: c) вычисления наибольшего общего делителя (НОД) двух положительных целых чисел a и b. Для построения алгоритма воспользуемся двумя свойствами НОД:

– если a = b, то D(a,b) = a;
– если a > b, то D(a,b) = D(a-b,b).

Следующее определение реализуется разбором случаев a = b, a<b и a>b:

```
D(nat a, nat b: nat c) ≡                                    (3.1)
        if a = b then c = a
        elsif a < b then D(a,b-a: c)
        else D(a-b,b: c)
        end
```

Каждый из вызовов D(a,b-a: c) и D(a-b,b: c) определяет хвостовую рекурсию и может быть заменен групповым оператором присваивания и переходом на начало определения. В групповом операторе значения аргументов вызова присваиваются входным параметрам определения.

```
D(nat a, nat b: nat c) ≡                                    (3.2)
    M:
        if a = b then c:=a
        elsif a < b then |a,b|:=|a,b-a|; goto M;
        else |a,b|:=|a-b,b|; goto M;
        end
```

Замена вызова предиката групповым оператором присваивания является эквивалентным преобразованием в соответствии с семантикой исполнения вызова предиката (см. п.2.5).

Цикл в определении (3.2) представим циклом общего вида:

loop ⟨оператор⟩ **end,**

в котором выход из цикла реализуется через оператор **exit**. Это преобразование не меняет процесса исполнения. Преобразование такого рода, улучшающее структуру программы, будем называть **оформлением**. Раскрывая также групповые операторы, получим итоговую императивную программу:

```
D(nat a, nat b: nat c) ≡
    loop
            if a = b then c:=a; exit
            elsif a < b then b:=b-a
            else a:=a-b
            end
    end;
```

3.3. Склеивание переменных

В результате проведения трансформаций в определении некоторого предиката часто становится возможной замена нескольких переменных одной. Это преобразование называется **склеиванием переменных**. Далее, при указании списка склеиваемых переменных условимся, что группа переменных заменяется первой переменной, указанной в списке. Склеивание массивов (и других структур) позволяет избежать их копирования, и поэтому является весьма эффективным преобразованием. Возможно также склеивание элементов одного массива в одной переменной.

Применение склеивания переменных возможно лишь при соблюдении **условий корректности**, гарантирующих эквивалентность исполнения по преобразованной программе. Эти условия предстоит разработать в дальнейшем. В рассматриваемых примерах правильность применения склеивания является очевидной.

3.4. Подстановка определения предиката

Подстановка определения предиката A(x: y) ≡ S вместо вызова A(e: z) в определении предиката B реализуется вставкой следующего блока вместо вызова. Сначала вставляется групповой оператор присваивания, реализующий присваивание аргументов вызова (набора выражений e) соответствующим параметрам набора x. Вслед за групповым оператором копируется оператор S — правая часть определения. Последним вставляется групповой оператор для пересылки значений набора y переменным набора z.

Если имена переменных для наборов x и y встречаются среди имен переменных в определении предиката B, то в общем случае необходимо предварительное систематическое переименование переменных в определении A(x: y) ≡ S, чтобы избежать возможных коллизий при подстановке определения.

Как правило, следующим после подстановки определения будет склеивание **родственных** переменных, которые до переименования были одинаково именованы в разных определениях; склеивание в подавляющем числе случаев возможно из-за отсутствия коллизий. Учитывая это, далее будем совмещать подстановку определения и склеивание родственных переменных в одном преобразовании, используя одно и то же имя для родственных переменных в разных определениях.

В случае подстановки определения предиката на место вызова функции соответствующий блок вставляется вместо вызова функции в позицию выражения. В результате получим конструкцию в стиле языка Алгол-68.

3.5. Обобщение исходной задачи

Это типичный прием предикатного программирования, когда мы переходим к другой, более общей, задаче для того, чтобы построить ее решение в виде хвостовой рекурсии, тогда как для исходной задачи подобное невозможно.

В качестве примера рассмотрим задачу Factorial(n: f) ≡ f = n! вычисления факториала от натурального значения n. На основе свойств 0! = 1 и n! = (n - 1)!n строится очевидная рекурсивная предикатная программа:

```
Factorial(nat n: nat f) ≡                                          (3.3)
        if n = 0 then f = 1 else f = n*Factorial(n-1) end
```

Рекурсия в этом определении не является хвостовой, поскольку вызов функции Factorial(n-1) не является последней исполняемой конструкцией — последней является операция умножения. Поэтому определение (3.3) нельзя преобразовать в цикл.

Обобщение исходной задачи строится таким образом, чтобы оно вбирало в себя последнее действие — умножение на n. Требуемой обобщающей задачей является fMult(n,m: f) ≡ f = n!m. Новым решением является следующая программа:

```
Factorial(nat n: nat f) ≡ fMult(n,1: f)                            (3.4)
fMult(nat n, nat m: nat f) ≡
                        if n = 0 then f = m else fMult(n-1,m*n: f) end
```

В этом решении рекурсия в определении fMult(n,m: f) имеет хвостовую форму, что позволяет заменить ее циклом.

```
fMult(nat n, nat m: nat f) ≡                                       (3.5)
        M: if n = 0 then f = m else |n,m| := |n-1,m*n| ; goto M end
```

Следующее преобразование — подстановка определения (3.5) в определение (3.4) для Factorial, причем подстановка совмещена со склеиванием родственных переменных n и f в определениях Factorial и fMult. Параметр m становится локальной переменной определения Factorial. Цикл в (3.5) оформим оператором **loop**.

```
Factorial(nat n: nat f) ≡
        nat m;
        |n,m|:=|n,1|;
        loop if n = 0 then f = m; exit else |n,m|:=|n-1,m*n| end end
```

Раскроем групповые операторы присваивания:

```
Factorial(nat n: nat f) ≡
        nat m;
        m:=1;
        loop if n = 0 then f = m; exit else m := m*n; n := n-1 end end
```

Последнее преобразование: склеивание переменных f и m. При замене цикла **loop** на **while** получим итоговую программу:

```
Factorial(nat n: nat f) ≡                                        (3.6)
                f:=1; while n ≠ 0 do f:=f*n; n:=n-1 end
```

3.6. Сравнение предикатного и императивного программирования

Эффективная программа (3.6) получена из предикатной программы (3.4) применением трех простых трансформаций. В традиционном императивном программировании программа (3.6) строится непосредственно с применением тех же свойств 0! = 1 и n! = (n-1)!n, и ее построение считается проявлением элементарной техники программирования. Доказательство правильности программы (3.6) реализуется индукцией по n.

В императивном программировании при разработке программы и ее модификации программист оперирует логическими свойствами программы, однако не в математическом стиле. Программа является плохим математическим объектом, и полноценная математическая работа с ней невозможна. Этот недостаток компенсируется отладкой и тестированием. Отметим, что многочисленные попытки автоматизировать традиционный стиль работы с программами оказались безуспешными.

Принципиальным отличием предикатного программирования является то, что все математические действия реализуются только на стадии решения, т.е. на математическом уровне, до получения окончательной предикат-

ной программы. Построение эффективной императивной программы из предикатной реализуется применением набора контролируемых трансформаций. Программа, модифицированная любой трансформацией, реализует процесс исполнения программы, эквивалентный процессу исполнения программы до ее модификации. Трансформации базируются на свойствах семантики исполнения, но не на математических свойствах программы. Любые математические действия с трансформируемой программой недопустимы.

Таким образом, **правильность** программы это правильность предикатной программы, что обеспечивается (или должно обеспечиваться) в рамках математики, а также правильность трансформаций и правильность их применения. **Эффективность** программы обеспечивается применением соответствующего набора трансформаций. Однако в большей степени эффективность программы зависит от выбора хорошего решения.

3.7. Иллюстрации на простых задачах

3.7.1. Вычисление суммы элементов массива

Рассмотрим задачу вычисления суммы Sum(n,X: s) элементов массива X длиной n>0, s обозначает значение суммы. Дадим описание типа массива X:

type Ar(**nat** k) = **array** 1..k **of real;** (3.7)

Это описание определяет тип вещественного массива, элементы которого имеют индексы 1,...,k, где k≥0 является параметром.

Для того чтобы получить решение задачи с хвостовой рекурсией, введем более общую задачу Sumk(n,X,k,w: s), где k≤n и

$$s = w + \sum_{i=k}^{n} X[i]$$

P-программа для задачи Sum состоит из двух определений:

Sum(**nat** n, Ar(n) X**: real** s) ≡ Sumk(n,X,1,0**:** s) (3.8)

```
Sumk(nat n, Ar(n) X, nat k, real w: real s) ≡                    (3.9)
        if k>n then s = w
        else Sumk(n,X,k+1,w+X[k]: s)
        end
```

Заменим хвостовую рекурсию в (3.9) циклом **loop**. Преобразованное определение (3.9) подставим в (3.8), склеивая родственные переменные и заменяя формальные параметры k и w локалами:

```
Sum(nat n, Ar(n) X: real s) ≡
        int k; real w;
        |k,w|:=|1,0|;
        loop
                if k>n then s = w; exit
                else |k,w|:=|k+1,w+X[k]|
                end
        end;
```

Раскроем групповые операторы и склеим переменные s и w:

```
Sum(nat n, Ar(n) X: real s) ≡
        int k;
        k:=1; s:=0;
        loop
                if k>n then exit
                else s:=s+X[k]; k:=k+1
                end
        end;
```

Наконец, заменяя цикл **loop** на **for,** получим итоговую программу:

```
Sum(nat n, Ar(n) X: real s) ≡                                    (3.10)
        int k;
        s:=0;
        for k:=1..n do s:=s+X[k] end;
```

Поскольку сумма элементов массива является часто встречающейся операцией, введем для суммы специальное обозначение: SUM(i = A..B, X[i]: s), где A и B — произвольные выражения.

Рассмотрим один способ распараллеливания вычисления суммы массива Sum(n,X: s). Допустим, n = p*m. Будем вычислять суммы элементов независимо по каждому диапазону индексов: k*m+1..(k+1)*m для k = 0,...,p-1, а затем суммировать p полученных сумм, которые будем накапливать в элементах массива Y:

```
type Ar0(nat k) = array 0..k of real;                                    (3.11)
sum(int p, int m, Ar(p*m) X: real s) ≡                                   (3.12)
        Ar0(p-1) Y;
        forAll k=0..p-1 do SUM(i = k*m+1..(k+1)*m,X[i]: Y[k]) end ;
        SUM(j = 0..p-1,Y[j]: s)
```

Императивная программа получается подстановкой (3.10) для каждого вхождения SUM в (3.12):

```
sum(int p, int m, Ar(p*n) X: real s) ≡
        int k,i,j; Ar0(p-1) Y;
        forAll k=0..p-1 do
                Y[k]:=0; for i:=k*m+1..(k+1)*m do Y[k]:=Y[k]+X[i] end
        end;
        s:=0; for j:=0..p-1 do s:=s+Y[j] end
```

3.7.2. Нахождение максимума массива

Рассмотрим задачу max(n,X: m,j) нахождения максимума в массиве чисел X длиной n>0; m обозначает значение максимума, а j — номер элемента, на котором достигается максимум, причем если таких элементов несколько, то выбирается тот из них, у которого наибольший индекс.

Решение реализуется просмотром элементов массива от конца к началу. Сначала устанавливаем начальные значения переменных m = X[n] и j = n, а затем при просмотре элементов уточняем эти значения. В соответствии с этой схемой рассмотрим обобщенную задачу maxS(n,X,i,ms,js: m,j), где m — максимум для значения ms и подмассива X с индексами 1..i(0<i<n). Если максимум достигается на ms, то j = js, иначе j есть индекс максимального элемента в X[1..i]. Р-программа представлена следующими двумя определениями:

```
max(nat n, Ar(n) X: real m, nat j) ≡ maxS(n,X,n-1,X[n],n: m,j)          (3.13)
maxS(nat n, Ar(n) X, nat i, real ms, nat js: real m, nat j) ≡           (3.14)
                if i = 0 then m = ms □ j = js
                elsif X[i]>ms then maxS(n,X,i-1,X[i],i: m,j)
                else maxS(n,X,i-1,ms,js: m,j)
                end
```

Императивная программа получается заменой рекурсивного определения (3.14) циклом и подстановкой его в правую часть (3.13), при этом формальные параметры i,ms и js становятся локалами:

```
max(nat n, Ar(n) X: real m, nat j) ≡
        nat js,i; real ms;
        |i,ms,js|:=|n-1,X[n],n|;
        loop
                if i=0 then m = ms □ j = js; exit
                elsif X[i]>ms then |i,ms,js|:=|i-1,X[i],i|
                else i:=i-1
                end
        end;
```

При раскрытии группового оператора |i,ms,js|:=|i-1,X[i],i| присваивание i:=i-1 следует поставить последним. Можно увидеть, что оператор i:=i-1 будет находится на ветвях **elsif** и **else**, его можно вынести за условный оператор. **Вынесение совпадающих вычислений** из ветвей условного оператора — это новый вид преобразований. Его применение в данном случае обусловлено раскрытием групповых операторов. Далее реализуется слияние пар переменных j и js, m и ms. Кроме того, "**nat** i; i:=n-1" заменяется на "**nat** i:=n-1". При замене цикла **loop** на **while** получим итоговую программу:

```
max(nat n, Ar(n) X: real m, nat j) ≡
        nat i:=n-1;
        m:= X[n]; j:=n;
        while i ≠ 0 do
                if X[i]>m then m:=X[i]; j:=i end;
                i:=i-1
        end
```

4. ПРОИЗВОДНЫЕ ФОРМЫ КОНСТРУКТОРА МАССИВА

Композиция (2.11) определяет общую форму конструктора массива. Пусть массив A есть переменная типа **array** I **of** T, причем тип I является строго упорядоченным множеством: $I = \{i_1, i_2, \ldots, i_n\}$. Рассмотрим следующую конкретизацию композиции (2.11):

```
Cons(x: A) ≡
    forAll i∈I do
        case i of
              i1: A[i] = E1(x)
            | i2: A[i] = E2(x) |...
            | in: A[i] = En(x)
        end
    end
```

Здесь x — набор переменных. Для предикатов $E_1(x: t)$, $E_2(x: t)$,..., $E_n(x: t)$ тип результирующей переменной t есть T. Приведенная композиция преобразуется в более простую форму:

$$Cons(x: A) \equiv A[i_1] = E_1(x) \square A[i_2] = E_2(x) \square \ldots \square A[i_n] = E_n(x)$$

Данная форма конструктора массива есть **поэлементное определение массива**. Будем использовать следующую форму записи:

$$A = \{E_1(x), E_2(x), \ldots, E_n(x)\} \qquad (4.1)$$

Рассмотрим другую форму конструктора массива: **объединение двух массивов**. Пусть имеются описания типов:

```
type ArA = array Ia of T;
type ArB = array Ib of T;
```

где типы Ia и Ib являются подмассивами некоторого одного типа и $Ia \cap Ib = \varnothing$. Определим тип индексов $I = Ia \cup Ib$ и тип массива:

```
type Ar = array I of T;
```

Определим предикат $\oplus$ как следующую конкретизацию композиции (2.11):

```
⊕(ArA a, ArB b: Ar c) ≡
        forAll i∈I do
            if i∈Ia then c[i] = a[i] else c[i] = b[i] end
        end
```

Заменим цикл **forAll** на два по типам Ia и Ib, затем проведем упрощения. Получим следующую форму конструктора в виде объединения массивов:

⊕(ArA a, ArB b: Ar c) ≡ (4.2)

forAll i∈Ia **do** c[i] = a[i] **end** □
forAll i∈Ib **do** c[i] = b[i] **end**

Далее будем использовать инфиксную форму: c = a⊕b. Операция ⊕ является коммутативной и ассоциативной.

Третья форма конструктора массива — **замещение элемента** — определяется как следующая конкретизация композиции (2.11):

Cons(Ar A, I j, T e: Ar B) ≡ (4.3)

forAll i∈I **do**
if i = j **then** B[i] = e **else** B[i] = A[i] **end**
end

Замещение элемента имеет следующее обозначение: B = A{j! e}. Если при реализации императивной программы массивы A и B будут склеены, то замещение элемента превратится в оператор: A[j]:=e.

Имеется еще одна форма конструктора массива: **вырезка массива**. Допустим, имеется другое описание типа массива:

type ArJ = array J of T;

где тип J⊂I. Вырезка массива определяется как следующая конкретизация композиции (2.11):

Cons(Ar A: ArJ B) ≡ **forAll** i∈J **do** B[i] = A[i] **end** (4.4)

Для вырезки будем использовать обозначение: B = A[J]. В частности, вырезка по диапазону n..m будет иметь вид B = A[n..m].

Подытожим определение производных форм конструктора массива:

⟨производные формы конструктора массива⟩ ::=
⟨поэлементное определение массива⟩ |
⟨объединение двух массивов⟩ |
⟨замещение элемента⟩ |
⟨вырезка массива⟩
⟨поэлементное определение массива⟩ ::=
[⟨изображение типа⟩]{⟨список выражений⟩}

Вхождение ⟨изображения типа⟩ необходимо в позициях, когда тип конструируемого массива определяется неоднозначно.

⟨объединение двух массивов⟩ ::= ⟨выражение⟩⊕⟨выражение⟩
⟨замещение элемента⟩ ::= ⟨выражение⟩{⟨выражение⟩ ! ⟨выражение⟩}
⟨вырезка массива⟩ ::= ⟨выражение⟩ [⟨изображение типа⟩]

В качестве иллюстрации рассмотрим алгоритм вычисления чисел Фиббоначи в соответствии с формулой:

$$f_1 = 1 \;\&\; f_2 = 1 \;\&\; (\forall k>2)(f_k = f_{k-1}+f_{k-2}) \qquad (4.5)$$

Предикат Fib(**nat** n**:** Ar(n) f) обозначает вычисление массива n чисел Фиббоначи, где:

type Ar(n) = **array** 1..n **of nat**;

Рассмотрим обобщение данной задачи в виде предиката

fibo(nat n, nat k, Ar(k-1) g**:** Ar(n) f) ,

вычисляющего требуемый массив f, используя массив g чисел Фиббоначи длины k-1, где $3 \le k \le n+1$. Тогда:

Fib(**nat** n**:** Ar(n) f) ≡ fibo(n,3,Ar(2){1,1}**:** f) (4.6)

Поэлементное определение массива {1,1} требует указания типа Ar(2). Определение предиката fibo использует определение типа одноэлементного массива:

```
type Ar1(j) = array {j} of nat;
fibo(nat n, nat k, Ar(k-1) g: Ar(n) f) ≡                    (4.7)
        if k=n+1 then f = g
        else fibo(n,k+1,g⊕Ar1(k){g[k-1]+g[k-2]}: f)
        end
```

Спецификация типа Ar1(k) является необходимой при определении массива {g[k-1]+g[k-2]}. Далее рассмотрим получение императивной программы. Заменим хвостовую рекурсию циклом:

```
fibo(nat n, nat k, Ar(k-1) g: Ar(n) f) ≡                    (4.8)
        loop
                if k = n+1 then f = g; exit
                else |n,k,g|:=|n,k+1,g⊕{g[k-1]+g[k-2]}|
                end
        end
```

Склеим переменные f и g. В результате этого массив g становится частью массива f и поэтому должен быть заменен вырезкой f[1:k-1]. При раскрытии группового оператора присваивания возникает оператор:

f[1:k]:=f[1:k-1]⊕{f[k-1]+f[k-2]}

Поскольку k-1 первых элементов левой и правой части совпадают, в итоге получим: f[k]:=f[k-1]+f[k-2]. Теперь подставим результат преобразования (4.8) в определение (4.6):

```
Fib(nat n: Ar(n) f) ≡                                          (4.9)
        k:=3; f[1]:=1; f[2]:=1;
        loop
                if k = n+1 then exit
                else f[k]:=f[k-1]+f[k-2]; k:=k+1
                end
        end
```

Последним преобразованием является замена цикла **loop** на **for**.

```
Fib(nat n: Ar(n) f) ≡                                          (4.10)
        f[1]:=1; f[2]:=1;
        for i = 3..n do f[i]:=f[i-1]+f[i-2] end
```

Имеет место парадоксальный факт: сходство программы (4.10), полученной в результате разнообразных преобразований, иногда нетривиальных, с начальным условием задачи (4.5). Фактически, формулу (4.5) можно считать вычислимой, причем программа (4.10) в точности описывает процесс ее вычисления.

Отметим, что схема решения данной задачи может быть использована для вычисления любых рекуррентных соотношений. Понятно, что форма (4.10) является предпочтительней, чем P-программа типа (4.6) и (4.7). Поэтому естественно включить форму (4.10) как часть императивного расширения языка P. При этом разумеется, композиция типа (4.5) не может быть базисной вычислимой логической композицией в языке исчисления предикатов. Итак, введем **рекуррентный оператор**:

⟨рекуррентный оператор⟩::=

⟨оператор⟩**; for** ⟨заголовок⟩ **do** ⟨оператор⟩ **end** |

⟨оператор⟩**; for** ⟨заголовок⟩ **do** ⟨элемент массива⟩=⟨выражение⟩ **end**

⟨заголовок⟩::=⟨переменная⟩=⟨выражение⟩..⟨выражение⟩ [**step** ⟨выражение⟩]

Если “**step** ⟨выражение⟩” отсутствует, шаг перевычисления ⟨переменной⟩ равен 1.

В качестве второго примера рассмотрим вычисление многочлена по схеме Горнера. Допустим, имеется многочлен степени $n \geq 0$:

$$u(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0$$

Вычисление многочлена представляется предикатом:

Su(**nat** n, Ar0(n) a, **real** x**: real** u) ,

где тип Ar0(n) определяется описанием:

type Ar0(**nat** k) = array 0..k **of real;**

Идея схемы Горнера определяется следующей формулой:

$$u(x) = (\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$$

Эта форма эксплицируется следующей рекуррентной схемой:

$$(v_0(x) = a_n) \& (\forall k=1\ldots n)(v_k(x) = v_{k-1}(x)x + a_{n-k}) \qquad (4.11)$$

Далее эта схема переписывается в виде программы с использованием рекуррентного оператора:

```
Su(int n, Ar0(n) a, real x: real u) ≡
        Ar0(n) v;
        v[0] =a [n]; for k = 1..n do v[k] = v[k-1]*x+a[n-k] end;
        u = v[n]
```

Для получения окончательной императивной программы достаточно склеить все элементы массива v с переменной u:

```
Su(int n, Ar0(n) a, real x: real u) ≡
                u = a[n]; for k=1...n do u:=u*x+a[n-k] end;
```

5. ГИПЕРФУНКЦИИ И РАСЩЕПЛЕНИЯ

Гиперфункция является адекватной формой спецификации многих программ. Понятия гиперфункции и расщепления определяются сначала на примере типовой ситуации для задачи решения системы линейных уравнений методом Гаусса. Далее иллюстрируются особенности решения этой

задачи, связанные с композицией нового вида “расщепление”. Наконец, в п. 5.3 дается полное формальное определение.

5.1. Определение для типовой ситуации

Рассмотрим задачу решения системы линейных уравнений:

$$\sum_{i=1}^{n} a_{ij} x_i = b_j,\ j = 1,\ldots,n \qquad (5.1)$$

Спецификацию задачи можно представить предикатом:

Lin(**nat** n, Matr(n) a, Ar(n) b**:** Ar(n) x) ,

где:

type Matr(**nat** k) = **array** {1..k; 1..k} **of real**; (5.2)

Описание (5.2) определяет массив с индексами в виде пары целых из диапазона 1..k. Запись {1..k; 1..k} является обозначением для изображения типа {(**nat** i,j)**:** $1 \le i \le k$ & $1 \le j \le k$}.

Для решения задачи Lin будем использовать алгоритм Гаусса (схему Жордана). Сначала алгоритм реализует исключение переменной x[1] для всех уравнений, кроме первого: из каждого i-го уравнения ($i \ne 1$) вычитается первое, помноженное на a[i,1]/a[1,1]. Аналогичным образом мы избавляемся от переменной x[2] во всех уравнениях, кроме второго. Повторяя эту процедуру для остальных переменных, получим систему уравнений с диагональной матрицей, решение которой очевидно.

Решение системы (5.1) может быть гарантировано при условии detNot0(a): детерминант матрицы a не равен 0. С учетом этого возможна следующая схема использования предиката Lin в некоторой программе:

... **if** detNot0(a) **then** Lin(n,a,b**:** x);A1(...) **else** A2(...) **end** ; ... (5.3)

Алгоритм Гаусса вполне возможно использовать также и для вычисления условия detNot0(a). Расширим спецификацию задачи Lin так, чтобы дополнительно вычислялось условие detNot0(a). Кроме того, в случае, когда детерминант равен нулю, будем дополнительно определять номер q переменной x[q], на которой обнаруживается вырождение системы (5.1), чтобы использовать эту информацию на ветви **else** в (5.3).

Если действовать в традиционном стиле императивного программирования, реализация предложенной идеи для схемы (5.3) могла бы выглядеть следующим образом:

... Lin(n,a,b: x,q,c); **if** c **then** A1(x...) **else** A2(q...) **end** ; ... (5.4),

где c — логическая переменная, значение которой есть detNot0(a). Отметим недостаток: одна из результирующих переменных x или q оказывается неопределенной.

В предикатном программировании предлагается кардинально новый подход.

Предикат Lin1(n,a,b: x) будет обозначать (5.1) при условии detNot0(a). Предикат Lin2(n,a,b: q) определяет первый номер q переменной x[q] вырождения системы (5.1), что возможно лишь при нарушении условия detNot0(a). Предикат Lin объединяет два предиката следующим образом:

$$(detNot0(a)\Rightarrow Lin1(n,a,b: x))\&(\neg detNot0(a)\Rightarrow Lin2(n,a,b: q)) \qquad (5.5)$$

Предикат (5.5) представляет склеивание двух функций, определяемых предикатами Lin1 и Lin2, в новую функцию, которая является ***гиперфункцией***, содержащей две ветви Lin1 и Lin2. В языке P гиперфункция (5.5) изображается следующим образом:

$$Lin(n,a,b: x|\ q) \qquad (5.6)$$

Гиперфункция есть новая форма спецификации задачи наряду с введенной ранее спецификацией в виде функции A(x: y). Спецификация задачи в виде гиперфункции содержит описание предиката (5.6) и условия detNot0(a), разделяющего области определения двух ветвей гиперфункции. Предикаты Lin1 и Lin2 называются ***проекциями*** предиката Lin соответственно на первую и вторую ветвь гиперфункции. Имеет место:

$$\begin{aligned} &Lin1(n,a,b: x) \equiv Lin(n,a,b: x|\ q)\&detNot0(a) \\ &Lin2(n,a,b: q) \equiv Lin(n,a,b: x|\ q)\&\neg detNot0(a) \end{aligned} \qquad (5.7)$$

Реализация схемы (5.3) представляется следующим предикатом:

$$\begin{aligned} &(detNot0(a)\Rightarrow Lin1(n,a,b: x)\&A1(x...))\& \\ &(\neg detNot0(a)\Rightarrow Lin2(n,a,b: q)\&A2(q...))... \end{aligned} \qquad (5.8)$$

Отметим, что для предикатов detNot0, Lin1 и Lin2 нет определений на языке P, поскольку они существуют как части предиката Lin(n,a,b: x| q), решение для которого будет дано ниже. Несмотря на внешнее сходство с

альтерацией, формула (5.8) есть базисная вычислимая логическая композиция нового вида, называемая "**расщеплением**" и представляемая на языке P следующим образом:

... **split** Lin(n,a,b**:** x| q) **do** 1**:** A1(x...) | 2**:** A2(q...) **end** ; ... (5.9)

Исполнение композиции "расщепление" реализуется следующим образом. При вычислении гиперфункции Lin реализуется выбор одной из двух ее ветвей с вычислением результирующих переменных на выбранной ветви. Если исполнение Lin завершилось на первой ветви, то далее вычисляется предикат A1(x...) после идентификатора ветви "1:". Если Lin завершилось на второй ветви, то вычисляется предикат A2(q...) после идентификатора "2:"**.**

Расщепление принципиально отличается от альтерации. В альтерации ветвление возникает по логическому значению предиката в первой позиции. В расщеплении ветвление реализуется внутри вычисления гиперфункции, причем завершение вычисления гиперфункции реализуется по одной из двух ветвей. В императивном расширении языка P гиперфункция реализуется в виде процедуры с двумя равноправными выходами. Конструкции такого рода до сих пор не использовались в существующих языках.

Расщепление (5.9) обладает также свойствами суперпозиции по каждой из двух ветвей, что видно из формулы (5.8). Существуют другие формы расщепления. Несколько упростим задачу Lin. Будем считать, что в случае вырождения матрицы a не требуется вычислять значение q, а достаточно лишь распознать факт вырождения. Спецификация задачи представляется гиперфункцией другого вида Lin(n,a,b: x|), где вторая ветвь не имеет результирующих переменных. Поскольку Lin2(n,a,b:) ≡ ¬detNot0(a), определение (5.5) переписывается для Lin(n,a,b: x|) следующим образом:

$$\text{detNot0(a)} \Rightarrow \text{Lin1(n,a,b: x)}$$

Реализация схемы (5.3) представляется формулой:

$$(\text{detNot0(a)} \Rightarrow \text{Lin1(n,a,b: x)}\&\text{A1(x...)})\&(\neg\text{detNot0(a)} \Rightarrow \text{A2(...)})...$$

и записываться на языке P как расщепление следующего вида:

... **split** Lin(n,a,b**:** x|) **do** 1**:** A1(x...) | 2**:** A2(...) **end** ; ...

Здесь суперпозиция имеется только на первой ветви; на второй реализуется альтерация.

5.2. Решение задачи с использованием гиперфункций и расщеплений

Вернемся к решению задачи Lin(n,a,b: x| q) по схеме Жордана. На ее примере покажем другие особенности. Рассмотрим более общую задачу Jord(n,a,b,k: x| q), где k = 1..n+1 и выполняется условие:

$$(\forall i=1..k-1)(a[i,i] = 1 \;\&\; (\forall j = 1..k-1)(i \neq j \Rightarrow a[i,j] = 0)),$$

т.е. матрица *a* является диагональной для первых k-1 переменных. Очевидным является следующее определение:

```
Lin(nat n, Matr(n) a, Ar(n) b: Ar(n) x | nat q)≡Jord(n,a,b,1: x | q)   (5.10)
```

При определении Jord будем использовать предикат trans(n,k,a,b: c,d), преобразующий систему уравнений (5.1) с матрицей *a* и вектором *b* в эквивалентную с матрицей *c* и вектором *d* так, чтобы c[k,k] = 1 и c[i,k] = 0 при i ≠ k. Заметим, что предикат trans применим лишь при условии a[k,k] ≠ 0. Если же a[k,k] = 0, то будем пытаться найти i-е уравнение с a[i,k] ≠ 0 (i>k) и поменять местами i-ю и k-ю строки в матрице a. Эти действия реализует предикат perm(n,a,b,k: c,d| q), причем вторая ветвь гиперфункции perm реализуется в случае, когда весь k-й столбец оказался нулевым, и тогда q = k.

В правой части определения Jord во всякой точке завершения исполнения будем использовать специальную конструкцию — **определитель ветви**: #1 или #2 — для указания того, какая из двух ветвей гиперфункции Jord реализуется в конкретной точке. Итак, дадим определение предиката Jord:

```
Jord(nat n, Matr(n) a, Ar(n) b, nat k: Ar(n) x | nat q) ≡                (5.11)
                if k = n+1 then x = b #1
                else split perm(n,a,b,k: c,d| q) do
                     | 1: trans(n,k,c,d: e,f);
                          split Jord(n,e,f,k+1:  x| q) do 1: #1 | 2: #2 end
                     | 2: #2
                     end
                end ;
```

Отметим важную особенность: определители ветвей #1 и #2 нужны лишь для реализации вычисления, они избыточны при рассмотрении правой части (5.11) как логической формулы.

Композиция **split** Jord(n,e,f,k+1: x| q) **do** 1: #1 | 2: #2 **end** является вырожденной и используется только для размещения определителей ветвей. Если ветвь расщепления содержит всего лишь определитель ветви, условимся втягивать определитель ветви в соответствующую позицию гипер-

функции, а саму ветвь удалять. С учетом этого правила определение (5.11) переписывается следующим образом:

```
Jord(nat n, Matr(n) a, Ar(n) b, nat k: Ar(n) x | nat q) ≡                (5.12)
                    if k = n+1 then x = b #1
                    else split perm(n,a,b,k: c,d| q #2) do
                            1: trans(n,k,c,d: e,f);
                                Jord(n,e,f,k+1: x #1| q #2)
                          end
                    end ;
```

Определение предиката trans слагается из преобразования матрицы *c* и вектора правой части *d*:

```
trans(nat n,k, Matr(n) c, Ar(n) d: Matr(n) e, Ar(n) f) ≡                 (5.13)
                    transf(n, k, c, d: f)  □
                    transe(n, k, c: e)
```

```
transf(nat n,k, Matr(n) c, Ar(n) d: Ar(n) f) ≡                           (5.14)
                    array {k} of real fk = {d[k]/c[k,k]};
                    array {1..k-1, k+1..n} of real ff;
                    forAll i = 1..k-1, k+1..n do ff[i] = d[i]-c[i,k]*fk[k] end;
                    f = ff ⊕ fk
```

В определении transf результирующий массив f строится в виде объединения двух массивов. Определение transe имеет похожую структуру.

```
transe(nat n,k, Matr(n) c: Matr(n) e) ≡                                  (5.15)
                    array {k; 1..n} of real ek;
                    forAll j = 1..n do
                        if j<k then ek[k,j] = c[k,j]
                        elsif j = k then ek[k,j] = 1
                        else ek[k,j] = c[k,j]/c[k,k]
                        end
                    end ;
                    array {1..k-1, k+1..n; 1..n} of real ee;
                    forAll i = 1..k-1, k+1..n do
                          forAll j = 1..n do
                                    if j<k then ee[i,j] = c[i,j]
                                    elsif j = k then ee[i,j] = 0
                                    else ee[i,j] = c[i,j]-c[i,k]*ek[k,j]
                                    end
                           end
                    end;
                    e = ee ⊕ ek
```

Приведем определение предиката perm:

```
perm(nat n, Matr(n) a, Ar(n) b, nat k: Matr(n) c, Ar(n) d|nat q)≡          (5.16)
                    if a[k,k] ≠ 0 then c = a □ d = b #1
                    else perm1(n,a,b,k,k+1: c,d #1| q #2)
                    end
```

Предикат perm1(n,a,b,k,m: c,d| q) определяет более общую задачу перестановки строк при дополнительном условии k<m&(∀i|k≤i<m)(a[i,k] = 0)).

```
perm1(n,a,b,k,m: c,d| q) ≡                                                 (5.17)
                  if m>n then q = k #2
                  elsif a[m,k] ≠ 0 then
                     {{ array {1..k-1, k+1..m-1,m+1..n} of real d1;
                        array {{1..k-1, k+1..m-1,m+1..n}; 1..n}, of real c1;
                        forAll i = 1..k-1, k+1..m-1,m+1..n do
                                    d1[i] = b[i]   □
                                    forAll j = 1..n do c1[i,j] = a[i,j] end
                        end
                      }□
                      {   array {k,m} of real d2 = {b[m], b[k]} □
                          array {{k,m}; 1..n}, of real c2;
                          forAll j = 1..n do
                                    c2[k,j] = a[m,j] □ c2[m,j] = a[k,j]
                           end
                      }
                     };
                     {d = d1 ⊕ d2 □ c = c1 ⊕ c2}
                     #1
                  else perm1(n,a,b,k,m+1: c,d #1| q #2)
                  end
```

При определении perm1 используется гибридная форма, реализующая совмещенный конструктор двух массивов c1 и d1.

Преобразование программы (5.10), (5.12)—(5.17) в императивную реализуется заменой рекурсии на цикл в Jord и perm1. Далее, подставим модифицированные тела Jord в (5.10), а perm1 в (5.16). Получим:

```
Lin(int n, Matr(n) a, Ar(n) b: Ar(n) x | int q) ≡                    (5.18)
        int k:=1;
        loop
                if k = n+1 then x = b #1 do
                else split perm(n,a,b,k: c,d| q #2)
                        1: trans(n,k,c,d: e,f);
                           |a,b,k|:=|e,f,k+1|
                        end
                end
        end
perm(nat n, Matr(n) a, Ar(n) b, nat k: Matr(n) c, Ar(n) d| nat q)≡   (5.19)
    if a[k,k] ≠ 0 then c = a □ d = b #1
    else
        int m:=k+1;
        loop
                if m>n then q = k #2
                elsif a[m,k] ≠ 0 then
                    {{  array {1..k-1, k+1..m-1, m+1..n} of real d1;
                        array {1..k-1, k+1..m-1, m+1..n; 1..n}, of real c1;
                        forAll i = 1..k-1, k+1..m-1, m+1..n do
                                d1[i] = b[i] □
                                forAll j = 1..n do c1[i,j] = a[i,j] end
                        end
                    } □
                    {   array {k,m} of real d2 = {b[m], b[k]} □
                        array {k,m; 1..n}, of real c2;
                        forAll j = 1..n do
                                c2[k,j] = a[m,j] □ c2[m,j] = a[k,j]
                        end
                    }
                    };
                    {d = d1 ⊕ d2 □ c = c1 ⊕ c2}
                    #1
                else m:=m+1
                end
        end
    end
```

Конструкции #1 и #2 интерпретируются в (5.18) и (5.19) как операторы завершения процедуры соответственно первым и вторым выходом. Подставим (5.14) и (5.15) в (5.13), затем (5.13) и (5.19) в (5.18), при этом выход #2 в perm становится выходом #2 в Lint, а выход #1 в perm выходит перед вызовом trans, так что второе вхождение #1 в perm можно заменить на **exit**.

Далее, совместим переменные для каждой из групп: a, c, c1, c2, e, ee, ek; x, d, d1, d2, b, f, ff, fk; q, k. В программе останется первая переменная из каждой группы. Учитывая, что вектор b — исходная переменная, а x — результирующая, вставим присваивание x:=b в начале программы. Как следствие, вместо **array** {k,m} **of real** d2 = {b[m], b[k]} возникает оператор |x[q],x[m]|:=|x[m],x[q]|, причем для раскрытия группового оператора необходима промежуточная переменная.

```
Lin(int n, Matr(n) a, Ar(n) b: Ar(n) x | int q) ≡
    x:=b;
    nat q:=1;
    loop
        if k = n+1 then #1 end;
        if a[q,q] = 0 then
          nat m:=q+1;
          loop
                if m>n then #2
                elsif a[m,q] ≠ 0 then
                  |x[q],x[m]|:=|x[m],x[q]| □
                  forAll j = 1..n do |a[q,j],a[m,q]|:=|a[m,j],a[q,j]| end ;
                  exit
                else m:=m+1
                end
          end
        end ;
        {   x[q]:=x[q]/a[q,q] □
            forAll j = q+1..n do a[q,j]:=a[q,j]/a[q,q] end
        };
        a[q,q]:=1;
        forAll i = q+1..n do x[i]:=x[i]-a[i,q]*x[q] end;
        forAll i = 1..q-1, q+1..n do
                forAll j = q+1..n do a[i,j]:=a[i,j]-a[i,q]*a[q,j] end □
                a[i,q] = 0
        end ;
        q:=q+1
    end
```

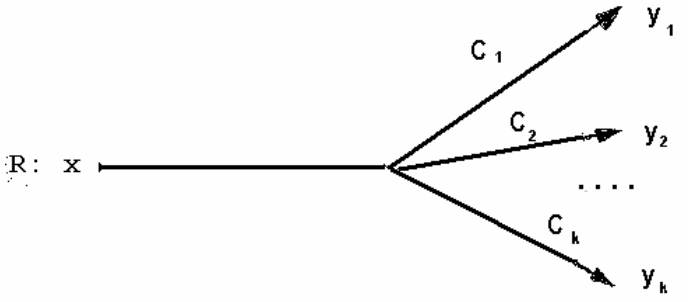
Раскрытие групповых операторов присваивания дает конечную императивную программу.

Можно обнаружить, что присваивания a[q,q]:=1 и a[i,q] = 0 являются избыточными, поскольку они не используются в вычислении вектора x. Кроме того, при обмене строк q и m достаточно обменивать элементы, начиная с номера q. Чтобы устранить эти излишние вычисления, следовало бы определить предикат Jord для вырезки a[q..n; 1..n] матрицы a.

5.3. Общее определение гиперфункции и расщепления

Рассмотрим спецификацию задачи в виде предиката $R(x: y_1,y_2,...,y_k)$, где $k>0$, $y_1,y_2,...,y_k$ — попарно непересекающиеся, возможно пустые, наборы переменных. С предикатом R связан также полный набор взаимно исключающих условий истинности ветвей. Это набор предикатов $C_1(x), C_2(x),...,C_k(x)$ таких, что $C_1(x)\vee C_2(x)\vee...\vee C_k(x)$ и $(\forall i,j)(i \neq j \Rightarrow (C_i(x)\&C_j(x)))$ — тождественно истинные формулы. В случае истинности условия $C_i(x)$ предикат R связывает x со значениями переменных набора y_i, тогда как значения переменных для других наборов $y_j(i\neq j)$ не определены.

Предикат $R(x: y_1,y_2,...,y_k)$ с набором условий $C_1(x), C_2(x),...,C_k(x)$ определяет **гиперфункцию** с k ветвями. Графически гиперфункция изображается следующим образом:



В языке P будем использовать следующую запись:

$R(x: y_1| y_2| ...| y_k)$

В случае $k = 1$ гиперфункция вырождается в обычную функцию. При $k>1$ гиперфункция представляет собой гибрид функции и распознавателя; ее вычисление реализует ветвление.

Вычисление гиперфункции реализуется следующим образом. В процессе вычисления выбирается ветвь i, для которой $C_i(x)$ — истинно. Исполнение гиперфункции завершается ветвью i. Вычисляются значения набора y_i.

Предикат $R^i(x{:}\ y_i)$ обозначает i-проекцию гиперфункции R на ветвь i:

$$R^i(x{:}\ y_i) \equiv R(x{:}\ y_1|y_2|...|y_k)\ \&\ C_i(x)$$

$R^i(x{:}\ y_i)$ является сужением предиката $R(x{:}\ y_1|\ y_2|\ ...|\ y_k)$ на область истинности условия $C_i(x)$ для входного набора x. Для непустого набора y_i i-проекция есть обычная функция, для пустого — $R^i(x{:}\ y_i) \equiv C_i(x)$.

Дадим общее определение для базисной вычислимой логической композиции **расщепление**. Расщепление есть композиция, определяющая гиперфункцию $R(x{:}\ y_1|\ y_2|\ ...|\ y_k)$, где $k>0$. Расщепление реализует ветвление с помощью другой гиперфункции $Q(v{:}\ z_1,t_1|\ z_2,t_2|\ ...|\ z_n,t_n)$, где $n>1$, $E_1(v)$, $E_2(v),...,E_n(v)$ — условия на ветви гиперфункции, наборы переменных $z_1,t_1,z_2,t_2,...,z_n,t_n$ могут быть пустыми. Композиция расщепления есть объединение независимо определяемых композиций для каждой из ветвей гиперфункции Q. Для произвольной ветви композиция может быть одной из следующих: суперпозицией, альтерацией, параллельной композицией или пустой композицией. Общий вид логической формулы расщепления есть конъюнкция n частей, которую для компактности представим через квантор $\forall$:

$$R(x{:}\ y_1|\ y_2|\ ...|\ y_k) \equiv \qquad (5.20)$$
$$(\forall i = 1..n)(E_i(v) \Rightarrow Q^i(v{:}\ z_i,t_i)\ \&\ A_i(u_i,z_i{:}\ e_1^i|\ e_2^i|\ ...|\ e_{si}^i))$$

где $Q^i(v{:}\ z_i,t_i)$ — проекция гиперфункции Q на i-ю ветвь, A_i — гиперфункция, используемая в композиции для i-й ветви, с числом ветвей $s_i \geq 1$.

Исполнение расщепления (5.20) реализуется следующим образом. Вычисляется гиперфункция $Q(v{:}\ z_1,t_1|\ z_2,t_2|\ ...|\ z_n,t_n)$. В результате выбирается некоторая ветвь i, и вычисляются наборы z_i,t_i. Далее вычисляется гиперфункция $A_i(u_i,z_i{:}\ e_1^i|\ e_2^i|\ ...|\ e_{si}^i)$. Допустим, ее результатом является ветвь m и набор переменных e_m^i. В этом месте исполнение (5.20) завершается. Набор e_m^i будет включен в число результирующих переменных предиката R.

Результатом исполнения (5.20) должна быть также указана ветвь гиперфункции R, которой завершается исполнение R. Для обозначения того, что j-ая ветвь — результат вычисления композиции, в позиции после результирующего набора e_m^i в гиперфункции A_i будем использовать конструкцию

определителя ветви #j, где j — целое, j = 1...k. Отметим, что определитель ветви требуется для вычислимости композиции (5.20) и избыточен при математических (логических) манипуляциях с формулой (5.20).

При изображении расщепления (5.20) на языке P в каждой m-ой позиции результатов всякой гиперфункции A_i вставляется определитель ветви $\#j_{im}$:

$R(x{:}\ y_1|\ y_2|\ \ldots|\ y_k) \equiv$
split $Q(v{:}\ z_1,t_1|\ z_2,t_2|\ \ldots|\ z_n,t_n)$ **do**
$1{:}\ A_1(u_1,z_1{:}\ e_1^{\,1}\ \#j_{11}|\ e_2^{\,1}\ \#j_{12}|\ \ldots|\ e_{s1}^{\,1}\ \#j_{1s1})$
$|\ 2{:}\ A_2(u_2,z_2{:}\ e_1^{\,2}\ \#j_{21}|\ e_2^{\,2}\ \#j_{22}|\ \ldots|\ e_{s2}^{\,2}\ \#j_{2s2})$
...
$|\ n{:}\ A_n(u_n,z_n{:}\ e_1^{\,n}\ \#j_{n1}|\ e_2^{\,n}\ \#j_{n2}|\ \ldots|\ e_{sn}^{\,n}\ \#j_{nsn})$
end

Если для i-й ветви $s_i = 1$, то вместо изображения "$|\ i{:}\ A_i(u_i,z_i{:}\ e_1^{\,I}\ \#j_{i1})$" будем использовать запись: "$|\ i{:}\ A_i(u_i,z_i{:}\ e_1^{\,i})\ \#j_{i1}$". Определитель ветви может быть опущен, если результирующей является первая ветвь.

Поскольку композиция для каждой ветви расщепления (5.20) строится независимо, рассмотрим произвольную i-ю ветвь расщепления:

$$E_i(v) \Rightarrow Q^i(v{:}\ z_i,t_i)\ \&\ A_i(u_i,z_i{:}\ e_1^{\,i}|\ e_2^{\,i}|\ \ldots|\ e_{si}^{\,i}) \qquad (5.21)$$

Здесь наборы v и u_i входят в набор x предиката R, а t_i и $e_j^{\,i}(j=1..s_i)$ являются результирующими для предиката R. Набор t_i может быть непустым при условии $s_i = 1$, т.е. если A_i вырождается в функцию.

Рассмотрим различные частные случаи для композиции (5.21).

Если набор z_i не пуст, то i-я ветвь является **суперпозицией**. Если набор z_i пуст, а t_i не пуст (и тогда $s_i = 1$), то имеет место **параллельная композиция**:

$$E_i(v) \Rightarrow Q^i(v{:}\ t_i)\ \&\ A_i(u_i{:}\ e_1^{\,i})$$

В принципе, вычисление предиката $A_i(u_i{:}\ e_1^{\,i})$ может происходить параллельно вычислению гиперфункции Q, однако если не будет выбрана i-я ветвь, то вычисление предиката A_i будет напрасным.

Далее, если оба наборы z_i и t_i пусты, композиция (5.21) принимает вид:

$$E_i(v) \Rightarrow A_i(u_i{:}\ e_1^{\,i}|\ e_2^{\,i}|\ \ldots|\ e_{si}^{\,i}) \qquad (5.22)$$

Композиция (5.22) является **альтерацией** по i-й ветви.

Композиция по i-й ветви (5.21) называется **пустой**, если предикат A_i является тождественно истинным, и тогда (5.21) принимает вид:

$$E_i(v) \Rightarrow Q^i(v\mathbf{:}\ t_i) \qquad (5.23)$$

Если в (5.23) набор t_i не пуст, то композиция по i-й ветви является **функцией**, а если t_i пуст, то **распознавателем**. Изображение i-й ветви (5.23) в языке Р принимает вид: "| i**:** #j_{i1}". Поскольку i-я ветвь фактически пуста, определитель ветви #j_{i1} переносится в i-ю ветвь вызова гиперфункции Q: Q(v**:** z_1,t_1| ...| t_i #j_{i1}| ...| z_n,t_n) в записи расщепления, а ветвь "| i**:** ..." удаляется.

Для расщепления с единственной непустой i-ой ветвью будем использовать иные формы записи:

Q(v**:** t_1 #j_{11}| ...| z_i,t_i| ...| t_n #j_{n1}); (5.24)
$A_i(u_i,z_i$**:** e_1^i #j_{i1}|e_2^i #j_{i2}| ...| e_{si}^i #j_{isi})

Q(v**:** t_1 #j_{11}| ...| t_i| ...| t_n #j_{n1}) ☐ (5.25)
$A_i(u_i$**:** e_1^i #j_{i1}| e_2^i #j_{i2}| ...| e_{si}^i #j_{isi})

Расщепление (5.24) определяет суперпозицию по i-й ветви. Расщепление (5.25) определяет по i-й ветви: параллельную композицию, если набор t_i не пуст, и альтерацию, если t_i пуст.

Например, для расщепления, встречающегося в определении (5.12):

```
split perm(n,a,b,k: c,d| q #2) do
| 1: trans(n,k,c,d: e,f);
     Jord(n,e,f,k+1: x #1| q #2)
end
```

применение формы (5.24) дает следуюшее изображение:

```
perm(n,a,b,k: c,d| q #2);
trans(n,k,c,d: e,f);
Jord(n,e,f,k+1: x #1| q #2)
```

Наконец, представим синтаксические правила для конструкций гиперфункции и расщепления. Начнем с изменений в структуре параметров определяющего вхождения предиката.

⟨описание исходных и результирующих параметров⟩::=
[⟨описания или обозначения параметров⟩]**:**
⟨описания или обозначения результатов⟩
⟨описания или обозначения результатов⟩::=
[⟨описания или обозначения параметров⟩]
[| ⟨описания или обозначения результатов⟩]

Зафиксируем изменения в структуре результатов ⟨вызова предиката⟩:

⟨результаты⟩::=
[⟨список переменных⟩] [⟨определитель ветви⟩]
[||⟨результаты⟩]
⟨определитель ветви⟩::= #⟨изображение натурального⟩
⟨оператор⟩::= … | ⟨определитель ветви⟩

Кроме того, любое вхождение оператора, кроме вхождений в блок, параллельный оператор и конструктор массива, может завершаться определителем ветви.

⟨оператор расщепления⟩::=
split ⟨вызов предиката⟩ **do** [||]⟨ветви расщепления⟩ **end**
⟨ветви расщепления⟩::=
⟨номер ветви⟩**:** ⟨оператор⟩
[|⟨ветви расщепления⟩]
⟨номер ветви⟩::=⟨изображение натурального⟩

6. СТРУКТУРНЫЕ ТИПЫ ДАННЫХ

Структурный тип определяется в виде композиции других типов, называемых **компонентными** по отношению к структурному. Структурными типами являются: массив (см. п. 2.6), кортеж, объединение и последовательность. Этот перечень, конечно, не исчерпывает всего многообразия структурных типов данных. Тип последовательности является фундаментальным типом, и поэтому он включен в приведенный набор наряду с другими. В принципе, его можно было бы определить через кортеж и объединение.

Представим правила описания структурных типов в языке Р.

⟨изображение структурного типа⟩::=
⟨изображение типа массива⟩|
⟨изображение типа кортежа⟩|
⟨изображение типа объединения⟩|
⟨изображение типа последовательности⟩|…

Изображение типа массива определено в п. 2.6.

⟨изображение типа кортежа⟩::=
struct ⟨изображение типов полей⟩ **end**
⟨изображение типов полей⟩::=
⟨изображение типа поля⟩ [⟨пробел⟩⟨имя поля⟩]
[,⟨изображение типов полей⟩]
⟨изображение типа поля⟩::=⟨изображение типа⟩
⟨имя поля⟩::=⟨имя⟩

Понятие типа и правила его изображения определены в п. 2.5.

⟨изображение типа объединения⟩::=
union ⟨изображение типов альтернатив⟩ **end**
⟨изображение типов альтернатив⟩::=
[⟨имя селектора⟩:]⟨изображение типа альтернативы⟩
[⊥⟨изображение типов альтернатив⟩]
⟨имя селектора⟩::=⟨имя⟩
⟨изображение типа альтернативы⟩::=[⟨изображение типа⟩]

Альтернатива является **пустой**, если после имени селектора отсутсвует ⟨изображение типа⟩.

⟨изображение типа последовательности⟩::=
seq ⟨изображение типа элемента⟩
⟨изображение типа элемента⟩::=⟨изображение типа⟩

Структурный тип определяется двумя **фундаментальными предикатами: Comp** и **Cons**. Предикат **Comp** связывает произвольное значение структурного типа с соответствующими значениями компонентных типов. Предикат **Cons**, называемый **конструктором**, по набору значений компонентных типов строит соответствующее значение структурного типа. Для каждого структурного типа вводятся специальные обозначения фундаментальных предикатов.

Для элемента массива используется обозначение ⟨переменная⟩ [⟨индекс⟩] (см. п. 2.6); имеется конструктор массива (оператор **forAll**) и его производные формы (см. п. 4).

Допустим, имеется тип R кортежа, состоящего из полей типов $T_1,T_2,...,T_n$: **type** R = **struct** T_1 f_1, T_2 f_2,..., T_n f_n **end**, где $f_1,f_2,...,f_n$ — имена полей. Дадим определения и обозначения предикатов Comp и Cons для кортежа r и полей $x_1,x_2,\ldots,x_n$:

$$\mathrm{Comp}(r\colon x_1,x_2,...,x_n) \equiv r\rightarrow(x_1,x_2,...,x_n) \quad (6.1)$$
$$\mathrm{Cons}(x_1,x_2,...,x_n\colon r) \equiv r = (x_1,x_2,...,x_n) \quad (6.2)$$

В соответствии с (6.1) и (6.2) тип R является произведением типов $T_1,T_2,...,T_n$. Обозначение $r.f_i$ используется для поля f_i кортежа r и соответствует значению i-го результата x_i предиката $\mathrm{Comp}(r\colon x_1,x_2,...,x_n)$: $r.f_i = \mathrm{Pr}(i,\mathrm{Comp}(r))$, где $\mathrm{Pr}(i,x)$ обозначает i-ю компоненту набора x.

Естественно соглашение, что для элемента массива вида A[(x,y)] используется традиционное написание A[x,y].

Допустим, имеется тип U объединения с типами альтернатив $T_1,T_2,...,T_n$: **type** U = **union** s_1:T_1| s_2:T_2| ...| s_n:T_n **end**, где $s_1,s_2,...,s_n$ — имена селекторов альтернатив. Определения предикатов Comp и Cons для объединения u и альтернатив $x_1,x_2,...,x_n$ следующие:

$$\mathrm{Comp}(u\colon x_1|\ x_2|\ ...|\ x_n) \equiv u\rightarrow(x_1|\ x_2|\ ...|\ x_n) \quad (6.3)$$
$$\mathrm{Cons}(s_i,x_i\colon u) \equiv u = (s_i\colon x_i),\ i = 1..n \quad (6.4)$$

Допустим, имеется тип S последовательности с элементами типа T:

type S = **seq** T.

Определения предикатов Comp и Cons для последовательностей *s*, *r* и элемента *e* следующие:

$$\mathrm{Comp}(s\colon |\ e,r) \equiv s\rightarrow(|\ e,r) \quad (6.5)$$
$$\mathrm{Cons}(\colon s) \equiv s = () \quad (6.6)$$
$$\mathrm{Cons}(e,r\colon s) \equiv s = (e,r) \quad (6.7)$$

Предикат (6.5) является утверждением, справедливым для произвольной последовательности s: s либо пуста (и тогда реализуется ветвь 1), либо для s имеется начальный элемент e и остаток последовательности r, причем реализуется ветвь 2.

Набор описаний типов может быть рекурсивным.

Приведем пример типа Tree2, определяющего регулярное двоичное дерево, каждая вершина которого содержит две дуги на поддеревья и атрибут типа T:

```
type Tree2 = union   s1: |
                     s2: struct T attr, Tree2 right, Tree2 left end
             end
```

Для атрибута *e* типа T, деревьев *r* и *l* типа Tree2 и кортежа *n* фундаментальным предикатом Comp для типа Tree2 является t→(|e,r,l). Он получается как результат подстановки n→(e,r,l) в t→(|n).

Введенные выше конструкции для структурных типов представим в синтаксических правилах.

```
⟨переменная⟩::=  ⟨простая переменная⟩ |
                 ⟨элемент массива⟩ |
                 ⟨поле переменной⟩
⟨поле переменной⟩::=⟨переменная⟩.⟨имя поля⟩
⟨первичное⟩::= … |
                 ⟨конструктор кортежа⟩ |
                 ⟨конструктор объединения⟩ |
                 ⟨конструктор последовательности⟩
⟨конструктор кортежа⟩::=
                 [⟨изображение типа⟩](⟨значения полей кортежа⟩)
⟨значения полей кортежа⟩::=
                 [⟨имя поля⟩:]⟨выражение⟩ [ , ⟨значения полей кортежа⟩]
⟨конструктор объединения⟩::=
                 [⟨изображение типа⟩](⟨имя селектора⟩:⟨выражение⟩)
⟨конструктор последовательности⟩::=
                 [⟨изображение типа⟩]() |
                 [⟨изображение типа⟩](⟨список выражений⟩) |
                 [⟨изображение типа⟩](⟨выражение⟩, ⟨выражение⟩)
```

Во втором случае ⟨список выражений⟩ определяют элементы последовательности. В третьем случае первым является ⟨выражение⟩ для элемента последовательности, а вторым — для продолжения последовательности.

⟨оператор⟩::= … |
⟨определение компонент кортежа⟩ |
⟨определение компонент объединения⟩ |
⟨определение компонент последовательности⟩
⟨определение компонент кортежа⟩::=
⟨имя переменной⟩→(⟨компоненты кортежа⟩)
⟨компоненты кортежа⟩::=
[⟨изображение типа⟩⟨пробел⟩]⟨имя переменной⟩
[,⟨компоненты кортежа⟩]
⟨определение компонент объединения⟩::=
⟨имя переменной⟩→(⟨компоненты объединения⟩)
⟨компоненты объединения⟩::=
⟨компоненты кортежа⟩ [⟨определитель ветви⟩]
[⊥⟨компоненты объединения⟩] |
[⟨определитель ветви⟩] [⊥⟨компоненты объединения⟩]

Второй случай правила реализуется для пустой альтернативы объединения.

⟨определение компонент последовательности⟩::=
⟨имя переменной⟩→([⟨определитель ветви⟩] |
[⟨изображение типа⟩⟨пробел⟩]⟨имя переменной⟩,
[⟨изображение типа⟩⟨пробел⟩]⟨имя переменной⟩
[⟨определитель ветви⟩])

Определение компонентной переменной действует в пределах суперпозиции данного оператора определения компонент с последующими операторами, причем определение компонент объединения или последовательности действует в пределах соответствующей ветви расщепления, базирующегося на данном операторе. Каждая из ветвей операторов определения компонент объединения или последовательности может завершиться определителем ветви.

7. ПРИМЕР. ВЫЧИСЛЕНИЕ СУММЫ ЧИСЕЛ В СТРОКЕ ЛИТЕР

Решение произвольной задачи по обработке последовательности реализуется в виде расщепления, инициируемого оператором определения компонент последовательности. Эту технику продемонстрируем на примере

простой задачи, которая была рассмотрена в [1] для иллюстрации трансформационного подхода к программированию.

Имеется строка литер, в которой встречаются числа в виде последовательности десятичных цифр. Числа разделяются произвольным набором литер, отличных от цифр. Подсчитать сумму чисел в строке.

Обозначим исходную задачу через SumString(String **s: nat** n), где s — строка литер, а n — сумма чисел в строке s. Тип String определяется следующим образом:

```
type String = seq char;                                    (7.1)
```

Введем более общую задачу Sum(String s, **nat** m: **nat** n), где n есть сумма чисел в строке плюс m. В отличие от SumString для предиката Sum можно построить рекурсивное определение в хвостовой форме. Итак:

```
SumString(s: n) ≡ Sum(s,0: n)                              (7.2)
```

Очевидным представляется следующий алгоритм. Сначала в строке определяется первая цифра. Далее вычисляется число, начинающееся с этой цифры. Для оставшейся части строки действие алгоритма повторяется. Построение предикатной программы реализуется непосредственной экспликацией этого содержательного описания алгоритма.

Определение первой цифры выражается гиперфункцией fDigit(s: | e,r), где первая ветвь фиксирует отсутствие цифр в строке, e — литера, являющаяся первой цифрой, r — оставшаяся часть строки после цифры. Отметим, что можно было бы использовать fDigit1(s: | r), где r начинается с первой цифры, однако в реализации это приведет к повторному доступу к одной и той же литере в строке, и поэтому использование fDigit предпочтительней.

Вычисление числа в строке выражается предикатом

```
stringNumber(e,r: v,t) ,
```

где *e* и *r* — те же, что и в fDigit, *v* — значение числа, *t* — оставшаяся часть строки после числа. Нетрудно доказать правильность следующего определения:

```
Sum(String s, nat m: nat n) ≡                              (7.3)
                    split fDigit(s: | e,r) do
                    | 1: n = m
                    | 2: stringNumber(e,r: v,t); Sum(t,m+v: n)
                    end
```

Рекурсивное определение fDigit строится следующим образом. Рассматривается оператор s→(| a,u), в котором на второй ветви строка *s* разбивается на литеру *a* и остаток строки *u*. Решение получается очевидным разбором случаев. Определение fDigit представлено как расщепление в форме (5.24):

```
fDigit(String s: | char e, String r) ≡                    (7.4)
                    s→(#1 | char a, String u);
                    if isDigit(a) then r = u □ e = a #2
                    else fDigit(u: #1| e,r #2)
                    end
```

Предикат isDigit(a) определяет, является ли литера *a* цифрой.

Чтобы реализовать предикат stringNumber через хвостовую рекурсию, необходим дополнительный параметр — накопитель. Рассмотрим предикат Number(r,p: v,t), где *r*, *v* и *t* — те же, что в stringNumber, а *p* — накопитель, равный значению разобранной части числа. Тогда получим:

```
stringNumber(char e, String r: nat v, String t) ≡         (7.5)
                    Number(r,numLit(e): v,t)
```

Функция numLit(e) вычисляет значение, определяемое литерой-цифрой *e*.

Справедливо следующее определение предиката Number:

```
Number(String r, nat p: nat v, String t) ≡                (7.6)
                   split r→( | char b, String q) do
                   | 1: v=p □ t=r
                   | 2:    if isDigit(b) then Number(q,p*10+numLit(b):
                   v,t)
                         else v = p □ t = q
                         end
                   end
```

Определения (7.1)—(7.6) образуют полную предикатную программу. Рассмотрим процесс преобразования Р-программы для получения эффективной императивной программы. Первым преобразованием является замена хвостовой рекурсии циклом в (7.3), (7.4) и (7.6). Приведем результат преобразования для (7.4):

```
fDigit(s: | e,r) ≡                                                    (7.7)
    loop
        s→( #1 | char a, String u);
        if isDigit(a) then r = u □ e = a #2
        else s:=u
        end
    end
```

Определители ветвей #1 и #2 оказываются здесь двумя различными выходами из цикла.

Чтобы получить полную императивную программу, подставим преобразованные фрагменты для (7.4) и (7.6) в (7.3):

```
SumString(s: n) ≡                                                     (7.8)
        nat m = 0;
        loop
            split
                loop
                    s→(#1 | char a, String u);
                    if isDigit(a) then {r = u □ e = a} #2 else s:=u end
                end;
            do
            | 1: n = m; exit;
            | 2: p:=numLit(e);
                loop
                        split r→( | char b, String q) do
                        | 1: {v = p □ t = r}; exit
                        | 2:  if isDigit(b) then r:=q; p:=p*10+numLit(b)
                              else {v = p □ t = q}; exit
                              end
                    end
                end;
                s:=t; m:=m+v
            end
        end
```

На следующем шаге реализуется совмещение переменных. Имеется несколько групп переменных: n,m; p,v; e,a; s,r,u,t,q. Переменные в каждой группе заменяются первой переменной. Для переменных последней группы, имеющих тип **seq char** в предикатной программе, предварительно необходимо выбрать конкретное представление. Рассмотрим тип:

```
type STRING = array 0..N of char;
```

Для последовательности s выберем конкретное представление в виде вырезки S[j..N] от массива S:

```
STRING S; int j = 0;
```

Тогда оператор s→(#1 | a, u) будет иметь следующую реализацию:

```
if j>N then #1 else a:=S[j]; j:=j+1 end
```

Применение этих преобразований к (7.8) дает программу:

```
STRING S; int j=0;                                                    (7.9)
N = 0;
loop
        loop
                if j>N then goto M1 else e:=S[j]; j:=j+1 end;
                if isDigit(e) then goto M2 end
        end;
    M1: exit;
    M2: p:=numLit(e);
        loop
                if j>N then exit else b:=S[j]; j:=j+1 end;
                if isDigit(b) then p:=p*10+numLit(b)
                else exit
                end
        end;
        n:=n+p
end
```

Вместо определителей #1 и #2 в (7.8) используются переходы по меткам M1 и M2 на разные ветви в объемлющем цикле.

Возможно другое конкретное представление для последовательности литер. Допустим, чтение очередной литеры из последовательности реализуется функцией readLit(), причем в случае исчерпания строки выдается значение EndOfString. Тогда оператор s→(#1 | a, u) будет реализован следующим образом:

```
a:=readLit(); if a=EndOfString then #1 else #2 end
```

Можно обнаружить следующий недостаток программы (7.9): если строка s завершается цифрой, то проверка исчерпания строки реализуется дважды. Чтобы исправить этот недостаток, следует использовать гиперфункцию stringNumber(e,r: v1| v2,t), в которой первая ветвь реализуется при исчерпании входной строки r. В этом случае определение (7.3) должно быть заменено на следующее:

```
Sum(s,m: n) ≡ split fDigit(s: | e,r) do                    (7.3')
              | 1:  n = m
              | 2:  split stringNumber(e,r: v1| v2,t) do
                   1: n = m+v1
                   2: Sum(t,m+v2: n)
                   end
              end
```

Расщепление является гибкой формой для выражения логики решения задачи. В существующих языках программирования проблематично адекватным образом записать программу (7.9), использование же **goto** общепризнано как дурной стиль. В такой ситуации либо запроцедуривается часть программы, соответствующая fDigit, либо вводится дополнительная логическая переменная, либо применяются другие "хитрости". Например, в [1] функция, аналогичная stringNumber, может работать также для пустой последовательности цифр. Это позволяет вызывать ее в начале тела рекурсивной процедуры Sum перед вызовом функции, аналогичной fDigit. Подобные программистские трюки демонстрируют несостоятельность традиционного программирования.

8. ПРИМЕР. РЕАЛИЗАЦИЯ БЫСТРОГО ПРЕОБРАЗОВАНИЯ ФУРЬЕ

Дискретное преобразование Фурье (ДПФ) вычисляет последовательность $\{y_k\}(k=0,\ldots,n-1)$ по исходной последовательности $\{x_m\}(m=0,\ldots,n-1)$ следующим образом:

$$y_k = \sum_{m=0}^{n-1} x_m W^{mk}, k = 0..n-1, W = \exp(-2\pi(\sqrt{-1})/n) \qquad (1)$$

Прямое вычисление по формуле (1) дает n^2 умножений. Ряд алгоритмов, ускоряющих вычисление ДПФ, получили название **быстрого преобразования Фурье** (БПФ). Рассмотрим алгоритм БПФ для $n = 2^t$ $(t>0)$, называемый "прореживанием по времени" [2], который сокращает число умноже-

ний до n*ln(n). При использовании свойств $W^n = 1$ и $W^{n/2} = -1$ формула (1) может быть преобразована к следующему виду:

$$y_k = \sum_{m=0}^{n/2-1} x_{2m} W^{2mk} + W^k \sum_{m=0}^{n/2-1} x_{2m+1} W^{2mk}, \qquad (2)$$

$$y_{k+n/2} = \sum_{m=0}^{n/2-1} x_{2m} W^{2mk} - W^k \sum_{m=0}^{n/2-1} x_{2m+1} W^{2mk}, k = 0..n/2-1$$

Если обозначить $W_n = W$, то очевидно, что $W_{n/2} = W^2$, и тогда каждая из четырех сумм в (2) есть ДПФ ранга n/2.

Исходя из схемы (2) сведения вычисления БПФ ранга n к вычислению двух БПФ ранга n/2, в работе [2] графически показано решение для t = 3 в виде так называемой "бабочки". Решение для произвольного t представлено как вычисление t шагов по следующей схеме:

$$i=1,...,t;\ l=1,...,n/2^i$$
$$x_l^i = x_l^{i-1} + W^d x_r^{i-1};\ x_r^i = x_l^{i-1} - W^d x_r^{i-1},$$

где $r = l+n/2^i$, d — инвертированое значение (см. ниже) для $l/2^{t-i}$.
По завершению t шагов $y_j = x_{j'}^t$;j=1,...,n, где j' — инвертированое значение j.

Данное решение, однако, не столь очевидно. Во вторых, оно дано практически в виде программы. Математическая часть решения, обосновывающая переход от схемы (2) к этому решению, отсутствует в изложении [2]. Поэтому определенная трудность заключается в воссоздании этой части решения.

Обозначим через DFT(**nat** n, ArC(n) x**:** ArC(n) y) задачу (1), где:

```
type ArC(nat k) = array 0..k-1 of complex;
```

Формула (2) может быть переписана в виде следующего определения:

```
DFT(nat n, ArC(n) x: ArC(n) y) ≡                                    (3)
    ArC(n/2) a; ArC(n/2) b;
    forAll k = 0..n/2-1 do a[k] = x[2*k] □ b[k] = x[2*k+1] end;
    ArC(n/2) A; ArC(n/2) B;
    {DFT(n/2,a: A) □ DFT(n/2,b: B)};
    forAll k = 0..n/2-1 do
            y[k] = A[k]+W**k*B[k] □ y[k+n/2] = A[k]-W**k*B[k]
    end
```

Чтобы вычисление определения (3) было эффективным, необходимо по меньшей мере избавиться от копирования массивов при их разбиении на две части и заменить рекурсию циклом. Последнее проблематично из-за наличия двух вхождений DFT в правой части. Типичный прием в такой ситуации — обобщить задачу (1) таким образом, чтобы два рекурсивных вхождения заменить одним.

Алгоритм (3) реализуется за t шагов. На нулевом шаге один массив длины n разбивается на два массива длины n/2. На i-м шаге имеем 2^i массивов длины $n/2^i$, получая при разбиении 2^{i+1} массивов. На шаге t-1 имеем n/2 массивов длины 2.

В определениях предикатов далее будем опускать параметры n и t, поскольку всюду они являются глобальными константами.

Рассмотрим задачу DF(ArC(n) x, **nat** i: ArC(n) y) вычисления ДПФ для 2^i массивов на i-ом шаге, причем эти массивы собраны в один массив *x* длины n. Результирующие 2^i массивов представлены в массиве *y* длины n. Приведем сначала определение DF(x,i: y) для i = 0, на его основе удобнее построить определение для произвольного i. Объединим массивы *a* и *b* из (3) в массив *d*, а массивы A и B — в массив *c*. Тогда на шаге 0 получим:

```
DFT(nat n, ArC(n) x: ArC(n) y) ≡ DF(x,0: y) ≡                    (4)
        ArC(n) d;
        forAll k = 0..n/2-1 do d[k] =x [2*k] || d[k+n/2] = x[2*k+1] end;
        ArC(n) c;
        DF(d,1: c);
        forAll k = 0..n/2-1 do
                y[k] = c[k]+W**k*c[k+n/2] ||
                y[k+n/2] = c[k]-W**k*c[k+n/2]
        end
```

Далее для задачи (1) будем использовать новое обозначение

FFT(ArC(n) x: ArC(n) y)

В дальнейшем, первая часть в суперпозиции в (4) будет обозначаться S(ArC(n) x, **nat** i: ArC(n) d), а третья — D(ArC(n) c, **nat** i: ArC(n) y). Приведем полное определение DF(x,i: y):

```
FFT(ArC(n) x: ArC(n) y) ≡ DF(x,0: y)                             (5)
```

```
DF(ArC(n) x, nat i: ArC(n) y) ≡                                    (6)
            if i = t then y = x
            else ArC(n) d; S(x,i: d); ArC(n) c; DF(d,i+1: c); D(c,i: y)
            end
S(ArC(n) x, nat i: ArC(n) d) ≡                                     (7)
      forAll l = 0..2**i-1 do
        forAll k = n*l/2**i..n*l/2**i-1+n/2**(i+1) do
                d[k] = x[2*k-n*l/2**i] □
                d[k+n/2**(i+1)] = x[2*k+1-n*l/2**i]
        end
      end
D(ArC(n) c, nat i: ArC(n) y) ≡                                     (8)
      forAll l = 0..2**i-1 do
            forAll k = n*l/2**i..n*l/2**i-1+n/2**(i+1) do
                  nat p = k*2**i;
                  { y[k] = c[k]+W**p*c[k+n/2**(i+1)] □
                    y[k+n/2**(i+1)] = c[k]-W**p*c[k+n/2**(i+1)]
                  }
            end
      end
```

В определении D(c,i: y) соответствующий множитель есть $(W_{n/2}i)^r$, где $r = k-n*l/2^i$. Используя свойство $W^{p+n*l} = W^p$, получим: $(W_{n/2}i)^r = W^p$, где $p = 2^ik$.

Анализируя исполнение определения (6), мы обнаруживаем, что в процессе рекурсивного исполнения (6) от шага i до шага t-1 вычисляется следующий терм:

$$S(...S(S(x,i),i+1),...,t-1) \qquad (9)$$

Вычисление терма (9) может быть вынесено из определения (6). Построим следующие определения:

```
In(ArC(n) x, nat i: ArC(n) u) ≡                                    (10)
            if i = t then u = x
            else ArC(n) d; S(x,i: d); In(d,i+1: u)
            end
HF(ArC(n) u, nat i: ArC(n) y) ≡                                    (11)
            if i = t then y = u
            else ArC(n) c; HF(u,i+1: c); D(c,i: y)
            end
```

Индукцией по i докажем, что имеет место тождество:

DF(ArC(n) x, **nat** i: ArC(n) y) ≡ In(x,i: u)**;** HF(u,i: y) (12)

Проверим определение (12) для i = t. Подставляя then-альтернативу из (6), а затем, используя then-альтернативы из (10) и (11), получим:

D(x,t: y) ≡ y = x ≡ u = x; y = u ≡ In(x,t: u); HF(u,t: y)

Допустим, (12) доказано для i+1 и требуется доказать для i. Справедлива следующая цепочка тождеств:

DF(x,i: y) ≡ S(x,i: d); DF(d,i+1: c); D(c,i: y) ≡ (13)
S(x,i: d); In(d,i+1: u); HF(u,i+1: y); D(c,i: y) ≡
In(x,i: u); HF(u,i: y)

Дадим пояснения. Поскольку i<t, подставляем else-альтернативу из (6) вместо DF(x,i: y). Далее, так как (6) доказано для i+1, подставляем правую часть (12) вместо DF(d,i+1: c). Получаем суперпозицию из четырех предикатов. В ней, поскольку i<t, первый и второй предикат соответствуют правой части (10) и поэтому заменяются левой частью (10). Аналогично, третий и четвертый предикат заменяются предикатом левой части из (11), что доказывает (12).

Преобразуем определение (11) к виду хвостовой рекурсии. Заметим, что (11) вычисляет следующий терм:

y = HF(u,i) = D(D(...D(u,t-1),t-2),...,i+1),i) (14)

Рассмотрим более общую задачу HL(e,i,j: y), вычисляющую терм

D(D(..D(e,j),j-1),..,i+1),i) ,

где j=t-1,...,i. Тогда:

HF(ArC(n) u, **nat** i**:** ArC(n) y) ≡ HL(u,i,t-1**:** y) (15)
HL(ArC(n) e, **nat** i, **nat** j**:** ArC(n) y) ≡ (16)
D(e,j**:** z); **if** j = i **then** y = z
else HL(z,i,j-1**:** y)
end

Упростим вычисление выражений 2^i, $n/2^{i+1}$ и других в определении D (8) с ориентацией на итеративное вхождение D в (16). Будем рассматривать более общую задачу D(c,i,f,r: y,g,v), где $f = 2^i$, $r = n/2^{i+1}$, $g = 2^{i-1}$, $v = n/2^i$, что позволит вычислять g = f/2 и v = r*2. В оптимизирующей трансляции такая

замена возведения в степень на умножение называется “уменьшение силы операций”. Обозначим $w_p = W^p$. Последовательность $\{w_p\}(p=0,...,n-1)$ может быть вычислена заранее. С учетом этого значение p в (8) должно вычисляться по модулю n. Итак:

```
D(ArC(n) c, nat i, nat f, nat r: ArC(n) y, nat g, nat v) ≡                (17)
    g = f/2 □  {nat v = r*2;
                forAll l = 0..f-1 do
                    nat o = l*v;
                    forAll k = o..o+r-1 do
                        nat p = f*k mod n; complex q = w[p]*c[k+r];
                        {y[k] = c[k]+q □ y[k+r] = c[k]-q}
                    end
                end
            }
```

Обобщение (17) для D требует аналогичного обобщения для HL в виде HL(e,i,j,f,r: y), где $f = 2^j$, $r = n/2^{j+1}$. Подставляя (12) в (5), получим:

```
FFT(ArC(n) x: ArC(n) y) ≡ In(x,0: u); HF(u,0: y)                          (18)
HF(ArC(n) u, 0: ArC(n) y) ≡ HL(u,0,t-1,n/2,1: y)                          (19)
HL(ArC(n) e, 0, nat j, nat f, nat r: ArC(n) y) ≡                          (20)
                    D(e,j,f,r: z,g,v);
                    if j = 0 then y = z
                    else HL(z,0,j-1,g,v: y)
                    end
```

В определении (20) условие j = 0 заменим эквивалентным f = 1. Поскольку параметр j становится ненужным также и в определении (17), обозначим HM(ArC(n) e, nat j, nat f, nat r: ArC(n) y) ≡ HL(e,0,j,f,r: y) и D’(c,f,r: y,g,v) ≡ D(c,i,f,r: y,g,v). Перепишем (19) и (20).

```
HF(ArC(n) u, 0: ArC(n) y) ≡ HM(u,n/2,1: y)                                (21)
HM(ArC(n) e, nat f, nat r: ArC(n) y) ≡                                    (22)
                    D’(e,f,r: z,g,v);
                    if f = 1 then y = z
                    else HM(z,g,v: y)
                    end
```

В итоге, определения (7, 10, 17, 18, 21, 22) представляют полную предикатную программу БПФ (1).

Далее рассмотрим реализацию императивной программы по предикатной. Подставим (17) в (22) с заменой рекурсии циклом, заменяя также единственное вхождение g на f/2:

```
HM(ArC(n) e, nat f, nat r: ArC(n) y) ≡                                  (23)
    loop
        { v = r*2;
          forAll l = 0..f-1 do
                o = l*v;
                forAll k = o..o+r-1 do
                      int p = f*k mod n; complex q = w[p]*e[k+r];
                      {z[k] = e[k]+q □ z[k+r] = e[k]-q}
                end
          end
        };
        if f = 1 then y = z; exit
        else |e,f,r|:=|z,f/2,v|
        end
    end
```

Принципиальным преобразованием является совмещение переменных *y*, *e*, *z* и *u* в одной переменной *y*, что устраняет операции копирования массивов. Вследствие совмещения, композиция {z[k] = e[k]+q □ z[k+r] = e[k]-q} должна быть заменена групповым оператором |y[k],y[k+r]|:=|y[k]+q,y[k]-q|. Для его раскрытия достаточно поменять присваивания местами: y[k+r]:= y[k]-q; y[k]:= y[k]+q.

Определение (10) не реализует существенных вычислений, а всего лишь "тасует" элементы массива x на каждом шаге. Известно [2], что преобразование In(x,0: y) реализует инвертацию массива x:

In(x,0: y) ≡ **forAll** i = 0..n-1 **do** y[i] = x[inv(i)] **end,** где функция inv(i) реализует двоичную инверсию целого числа, представленного двоичным изображением длины t. В результате операции биты значения i переписываются в обратном порядке — инвертируются. Например, для n = 8 значения 0, 1, 2, 3, 4, 5, 6, 7 инвертируются, соответственно, в значения 0, 4, 2, 6, 1, 5, 3, 7.

Приведем полную императивную программу:

```
w[0] = 1; for i = 0..n-2 do w[i+1] = w[i]*W end ;                    (24)
forAll i = 0..n-1 do y[i] = x[inv(i)] end;
f:=n/2; r:=1;
loop
  { v = r*2;
    forAll l = 0..f-1 do
      o = l*v;
      forAll k = o..o+r-1 do
            p = f*k mod n; q = w[p]*y[k+r];
            y[k+r]:=y[k]-q; y[k]:=y[k]+q
      end
    end
  };
  if f = 1 then exit
  else f:=f/2; r:=v
  end
end
```

Главное отличие программы (24) от предложенной в [2] состоит в том, что в [2] исходный массив не инвертируется, однако результирующий массив имеет инвертированное представление. В принципе, алгоритм в [2] можно получить преобразованиями представленной выше предикатной программы, для чего следует использовать следующее свойство инвертации: inv(inv(i)) = i. Можно показать, что в случае применения второй инвертации к исходному массиву картина дробления массивов от шага 0 до шага t-1 станет такой же, как в программе (24) от шага t-1 до шага 0.

Программу (24) можно закодировать на любом императивном языке, и она будет достаточно эффективной. Однако она не оптимальна и, например, будет уступать программе, полученной Фурье-транслятором [3]. Имеется много способов улучшить полученную предикатную программу. В частности, можно определять ситуации, когда $w_p = 1$, и реализовывать вычисления без умножения на w_p. Для нас принципиально то, что любое из известных улучшений можно провести в рамках предикатного программирования.

Использование рекурсивных определений неизбежно почти в любой предикатной программе. В первую очередь, возникает вопрос корректности рекурсии, что обычно отождествляется с ее завершимостью. Далее необходима разработка общих правил по технике преобразования рекурсии. Например, реализовано вынесение предиката S(x,i: d) из тела рекурсивного определения (6):

```
DF(x,i: y) ≡    if i = t then y = x                                    (6)
                else S(x,i: d); DF(d,i+1: c); D(c,i: y)
                end
```

Это было вынесение перед вхождением DF. Доказана правильность вынесения, причем доказательство зависит не от предикатов S, DF и D, а зависит всего лишь от формы композиции (6). Следовательно, можно сформулировать более общие условия вынесения некоторой части из рекурсивного определения. Заметим, что в композиции (6) аналогичным образом можно было бы вынести предикат D(c,i: y) из определения за вхождение DF.

9. ОБЗОР СМЕЖНЫХ ПОДХОДОВ

Общепризнанными являются следующие формализации понятия программы: машина Тьюринга, частично рекурсивные функции, нормальный алгорифм Маркова [6] и каноническая система Поста [7]. Известны также модели программ: схема программы в теории схем программ [8], сеть Петри [9] и др. Данный фундамент, однако, существует обособленно и фактически слабо связан с реальным программированием. В настоящей работе предложена иная формализация понятия программы: исчисление вычислимых предикатов. Данная модель программы с достаточной полнотой отражает множество программ, спецификация которых представима в виде предиката.

Предикатное программирование можно считать развитием функционального программирования. Существует много общего между предикатными и функциональными программами. Главное их сходство — они являются правильными математическими конструкциями. Проведем сравнение по различным видам конструкций.

В языках функционального программирования нет прямого аналога параллельной композиции, поскольку она определена для предикатов и подразумевает наличие нескольких результирующих переменных. Наиболее распространен стиль с одной результирующей переменной. Например, язык ML [11] признает наличие более одной результирующей переменной лишь при определении функций высших порядков. Вероятно, противоположным этому стилю является язык Sisal 90 [12], ориентированный на написание параллельных программ. В нем имеется конструкция ⟨список выражений⟩, предполагающая независимое параллельное вычисление каждого из выражений списка.

Использование массивов нетипично для языков функционального программирования. Вместо массивов обычно используются списки и деревья. В языке ML разрешены только одномерные массивы константной длины; многомерные массивы трактуются как массивы, элементами которых являются массивы. В языке Sisal 90 [12], напротив, разрешены лишь динамические массивы, т.е. массивы, для которых верхняя граница индексов является переменной. В качестве индексов допускается лишь диапазон целых чисел. Многомерный массив трактуется как массив массивов. В описании типа массива верхняя граница индексов не фиксируется. В языке Модула-2 [13] подобные массивы называются открытыми.

Массивы, подобные определяемым в языке P, допустимы в языке функционального программирования ALPHA [21]. В качестве типа индексов многомерного массива в языке ALPHA используется произвольный многогранник, задаваемый системой линейных неравенств. Как и в языке P, тип индексов, а значит и тип массива, может быть параметризован переменными. Напомним, что в языке P в качестве типа индексов допустим произвольный конечный тип.

Аналог конструктора массива (конструкция **forAll**) имеется в языке FORTRAN 90. В языке Sisal 90 нет подобной конструкции. Работа с массивами там полностью динамизирована. Используются потоки выражений, длина которых зависит от вычислений. Вычисление потока значений и присваивание значений потока элементам массива разъединено и оформлено в Sisal 90 двумя различными конструкциями. Производные формы конструктора массива встречаются в языках Sisal 90 и ML. В языке ALPHA [21] в роли конструктора одного или нескольких массивов используется конечный набор афинных рекуррентных уравнений, подобных определяемым в линейном программировании.

Структурные типы данных определены во многих языках функционального программирования. В языке Lisp [10] рекурсивные структуры данных могут быть построены в виде списков, в языке ML — в виде деревьев. Тип объединения используется для построения рекурсивных типов данных в аппликативном языке [5]. Аналогами последовательностей языка P могут быть списки языка ML.

Для гиперфункции и расщепления нет прямых аналогов ни в функциональных, ни в императивных языках программирования. В существующих языках, возможно, наиболее близкой к расщеплению является следующая комбинация с использованием результирующей переменной z типа объединения и оператора **case** специального вида:

```
z:= ⟨...⟩; case tag z of
            | tag1: z = ⟨поля альтернативы 1⟩; ⟨оператор1⟩
            | ...
            | tagN: z = ⟨поля альтернативы N⟩; ⟨операторN⟩
          end
```

Каждая альтернатива оператора **case** определена для соответствующей альтернативы структуры z. В начале альтернативы **case** переменная z отождествляется со списком переменных, представленных полями данной альтернативы объединения. Подобный оператор **case** есть, например, в языке Sisal 90.

Понятие хвостовой (tail) рекурсии используется во многих языках функционального программирования (например, в [10,11,14]) как рекурсии, которая может быть заменена циклом при трансляции функциональной программы. Возможность обобщения определяемой функции для достижения хвостовой рекурсии также показана в этих языках. Замена рекурсии циклом рассматривалась там как оптимизация в процессе трансляции функциональной программы. Понятно, что одна эта оптимизация не обеспечивает принципиального улучшения эффективности транслируемой программы. Напротив, в предикатном программировании преобразование по замене рекурсии циклом приобретает особую значимость, поскольку это преобразование оказывается стартовым для цепочки трансформаций, обеспечивающих получение эффективной императивной программы.

Почти любой язык функционального программирования имеет **императивное расширение** — дополнительный набор языковых конструкций, свойственных императивным языкам программирования. Эти конструкции могут использоваться для улучшения эффективности функциональной программы. При этом декларируется, что использование императивных конструкций должно быть ограниченным и не нарушать прозрачности функциональной программы. Исключением этому правилу является язык Sisal 90. В нем разрешено регулярное использование циклов типа **while** и **repeat**, операторов присваивания вида `x := old x + 1` в телах циклов (где **old** x — значение переменной x до присваивания) и других конструкций. Поэтому, формально, язык Sisal 90 нельзя считать языком функционального программирования. Однако с точки зрения возможностей анализа принципиально, что Sisal-программа имеет правильную SSA (static single assignment) форму [15].

Возможен другой подход в соединении функциональной и императивной частей языка. Язык FISh [16,17] определен как смесь функциональных

и императивных конструкций. Он позволяет использовать при построении программы полиморфные функции высших порядков в стиле языка ML и реализацию функций в виде императивных процедур. Для обеспечения эффективности реализовано частичное исполнение функциональной части FISh-программы с преобразованием ее в императивные конструкции и последующей конвертацией итоговой программы на язык C. При частичном исполнении вычисляются атрибуты типов (type inference в стиле трансляции с языка ML), в том числе для переменных типа “массив”. Для этой цели в языке FISh имеются переменные типа Shape, и в результате частичного исполнения реализуется вычисление значений таких переменных. Вычисленная информация о типах дает возможность эффективной реализации открытой подстановки тел функций на места их вызовов, развертки ограниченной рекурсии, замены доступа через указатель на прямой доступ к объекту (unboxing) и других преобразований, обеспечивающих эффективность конечной программы.

С предикатным программированием связано также другое направление — трансформационное программирование, известным представителем которого является проект CIP [4, 5, 1]. В проекте CIP определяется язык CIP-L [4], называемый также “The wide spectrum language 84” [5]. Это язык, содержащий следующие подъязыки:

- язык спецификаций, аналогичный языку исчисления предикатов;
- аппликативный язык — язык функционального программирования;
- диалекты императивных языков Pascal и Algol, причем Algol существенно отличается от Algol-60 и Algol-68; в частности допускаются операторы перехода **goto.**

CIP-L определяет структурные типы данных (без массивов), в том числе рекурсивно определяемые типы, и абстрактные типы данных с описанием свойств операций типов и с возможностью инкапсуляции представления значений типов. Различные уровни языка CIP-L интегрируются формальным описанием трансформационной семантики. Трансформационное правило для каждой языковой конструкции определяет построение математической функции из функций для подконструкций данной конструкции.

Построение программы на языке CIP-L реализуется применением набора трансформаций для начальной более простой версии программы. Процесс трансформации поддерживается системой CIP-S, которая для каждой применяемой трансформации проверяет условия ее применимости, а также автоматически реализует саму трансформацию.

Аппарат трансформаций является универсальным в CIP. Трансформации могут применяться для доказательства утверждений, для преобразования программы в рамках аппликативного языка, для преобразования с аппликативного языка на императивный, а также для оптимизации программы на императивном языке. Имеются трансформации, преобразующие так называемую линейную рекурсию в хвостовую, хвостовую рекурсию — в цикл типа WHILE.

Каталог трансформаций CIP-S включает также трансформационные правила для конструкций языка CIP-L. Подчеркнем, что трансформационные правила фактически базируются на денотационной семантике.

Система CIP оказалась достаточно сложной для использования в производственном программировании.

В отличие от CIP, трансформации не применяются на уровне языка предикатного программирования, а преобразование рекурсии к хвостовому виду является частью техники предикатного программирования и реализуется как часть процесса решения задачи. Трансформации применяются только для преобразования предикатной программы в императивную, причем все они базируются не на денотационной, а на операционной семантике.

Имеется ряд работ по трансформации функциональных программ для получения эффективных параллельных программ. Существует подход [18] по трансформации функциональных программ в эффективные сети последовательных взаимодействующих процессов Хоора (CSP) [19]. Предлагается стратегия трансформации функциональных программ для алгоритма Divide_and_Conque с неявно выраженным параллелизмом в функциональные программы с явным параллелизмом [20].

Разработана методология трансляции функциональной программы на языке ALPHA [21,22,23] в эффективную параллельную императивную программу. В рамках языка ALPHA применяются следующие трансформации: изменение базиса индексов массива, подстановка и нормализация. На базе проводимого статического анализа программы в процессе трансляции реализуется серия трансформаций программы: выравнивание переменных, разделение и физическое размещение, генерация системы циклов и др. [22]. Данная методология трансляции адаптирована также для синтеза интегральных схем процессора, реализующего регулярные операции с массивами, на примере процессора полиномиального деления [23].

ЗАКЛЮЧЕНИЕ

Для класса программ, спецификация которых представима в виде математического предиката, предложена формализация понятия программы в виде исчисления вычислимых предикатов. Исчисление строится на основе семи видов базисных вычислимых логических композиций. Нет уверенности в полноте этого набора видов, и вполне вероятно появление в будущем других видов композиций в исчислении. Однако существующего базиса пока достаточно для видимого спектра возможных приложений.

Язык предикатного программирования систематическим образом построен из исчисления вычислимых предикатов. Все языковые конструкции, за исключением связанных с типами данных, получены из базисных вычислимых логических композиций. Выбор той или иной формы определялся также рядом общеизвестных требований на язык программирования. Исследованы свойства суперпозиции и параллельной композиции для обоснования правил опускания скобок и легализации традиционного функционального стиля, применяемого в языках программирования. Выражения были введены как производная форма суперпозиции с использованием функционального стиля и инфиксной формы для примитивных предикатов. Конструктор массива представлен универсальной формой (2.10). Показано, что все другие виды конструктора массива являются его производными формами. Очевидно, что средства работы с массивами будут в дальнейшем существенно развиваться.

Отличительной особенностью языка предикатного программирования является возможность использования произвольного конечного типа в качестве типа индексов массива. В связи с этим возникает необходимость определения типа подмножества как области истинности предиката. В итоге возникают типы, параметризованные переменными. Может показаться, что эти свойства языка не являются обязательными и можно было бы ограничиться более простой конструкцией массивов, как например, в языке Sisal 90 [12]. Можно обнаружить, что указанные свойства языка предопределены требованием универсальности базисной композиции (2.10) для конструктора массива. Поскольку конструктор определяет массив для всех своих индексов, то более широкий тип недопустим в качестве типа индексов этого массива. Следовательно, тип индексов должен быть строго задан с точностью до элемента, и в принципе, может быть произвольным.

Таким образом, исчисление вычислимых предикатов является адекватным методологическим базисом при построении языка предикатного программирования. В этом плане показательным является отрицательное ре-

шение по вопросу включения рекуррентного оператора в качестве нового вида базисных вычислимых логических композиций. Рекуррентная формула (4.5) для чисел Фиббоначи является правильной логической и вычислимой формулой. Тем не менее, для любых рекуррентных соотношений типа (4.5) можно определить решение на языке P, из чего следует, что формула типа (4.5) не является базисной.

Предикатное программирование принципиально шире функционального по ряду позиций. Параллелизм предикатной программы представлен параллельной композицией явным образом. Используется более общая форма суперпозиции (2.1) по сравнению с функциональными языками. Средства работы с массивами предоставляют значительно большие возможности описания алгоритмов. Например, умножение матриц будет представлено в форме, близкой к традиционной математической, тогда как в функциональном языке (например, ML) потребуется использование целого набора функций. Наконец, имеются гиперфункция и расщепление, которые заведомо невозможно определить в функциональных языках.

Императивная программа является плохим математическим объектом, и полноценная математическая работа с ней принципиально невозможна. Этот факт является главной причиной невозможности обеспечения надежности программ в рамках традиционного императивного программирования и, следовательно, его бесперспективности. Предикатное программирование — это подход, в котором все математические действия с программой реализуются в рамках предикатной программы, т.е. на математическом уровне. При получении соответствующей императивной программы любые математические операции с трансформируемой программой недопустимы. По этой причине, например, такая оптимизация, как уменьшение силы операций (с заменой умножения на сложение), может проводиться только при построении предикатной программы.

Итак, правильность программы — это правильность предикатной программы, а также трансформаций и их применения. Правильность предикатной программы должна обеспечиваться средствами ее автоматической верификации с применением всего математического аппарата. Эффективность программы обеспечивается применением соответствующего набора трансформаций. При этом в большей степени эффективность программы зависит от выбора хорошего решения.

Процесс получения эффективной императивной программы из предикатной определен как последовательность применения трансформаций трех видов: замены хвостовой рекурсии циклом, подстановки определения предиката на место его вызова и склеивания переменных. Склеивание пере-

менных является менее очевидной трансформацией. Для массивов возможно склеивание массивов разной длины (см. п. 4); аналогичные особенности возможны для структурных переменных. Кроме того, не определены критерии применимости (правильности) склеивания переменных. Первоначально необходимо разработать классификацию разных способов склеивания структурных переменных, а затем дать формальное определение каждого из способов с описанием критериев применимости. Приведенный набор из трансформаций трех видов, разумеется, неполон. Например, в п. 3.7.2 используется трансформация — вынесение совпадающих вычислений из ветвей условного оператора, эти совпадающие вычисления возникли при раскрытии групповых операторов. Вне рассмотрения остались различные способы представления структурных переменных. Разработка полной классификации трансформаций является одной из дальнейших задач.

Предикатное программирование может быть поддержано системой программирования со следующими функциями:

- автоматизированная верификация предикатной программы на соответствие спецификации, а также контроль семантики конструкций языка P;
- поддержка применения трансформаций программы;
- трансляция итоговой программы в объектный код или конвертация в один из языков программирования: C, Фортран и др.

В отличие от языков функционального программирования параллелизм предикатной программы представлен явным образом посредством параллельной композиции и конструктора массива (цикла **forAll**). Этот параллелизм является максимальным; в результате трансформации предикатной программы параллелизм может быть лишь ограничен. В параллельной предикатной программе нет конфликтов по доступу, однако конфликты могут появиться в результате трансформаций. Предикатные программы в большей степени подходят для их реализации на параллельных вычислительных системах, чем программы на существующих императивных языках. Реализация параллелизма — это открытая область исследований в предикатном программировании.

Гиперфункция оказывается адекватной формой спецификации многих программ, в особенности для системных задач. Оператор расщепления, конструируемый на базе гиперфункции, определяет гибкие формы, позволяющие свободно и адекватно отразить любое развитие предикатной программы. Обработка исключений, например, подобных исключениям языка Java, естественным образом выражается посредством операторов расщеп-

ления и не требует включения в язык Р дополнительных специальных языковых средств. Гиперфункции и расщеплению нет аналогов ни в одном языке программирования.

Реализацией гиперфункции в императивном расширении языка Р является процедура с несколькими равноправными выходами, если только определение гиперфункции не подставляется открыто. Отметим, что реализация процедур с несколькими выходами должна быть существенно проще и эффективней реализации механизма обработки исключений для существующих языков программирования. Процедур с несколькими выходами еще не было в практических языках программирования.

Автор благодарен А.Г.Марчуку за множество критических замечаний и поддержку работы. Автор признателен В.А.Непомнящему за интерес к работе и полезные критические замечания. Автор благодарен Н.В.Шилову и А.А.Можейко за предоставленные материалы и обсуждение работы.

СПИСОК ЛИТЕРАТУРЫ

1. **Brass B., Erhard F., Horsch A., Riethmayer H.-O., Steinbruggen R.** CIP-S: An instrument for program transformation and rule generation. — Munchen, 1982. — P. 44–62. — (Prepr. / Inst. For Informatik, Technische Universitat Munchen; TUM-18211)
2. **Нуссбаумер Г.** Быстрое преобразование Фурье и алгоритмы вычисления сверток / Пер. с англ. — М.: Радио и связь, 1985. — С. 75–91.
3. **Frigo M.** A Fast Fourier Transform Compiler // ACM SIGPLAN Notices. — 1999. — Vol.4, N.5. — P.169–180.
4. **Bauer F.L., Broy M., et.al.** The Wide Spectrum Language 84. — Inst. For Informatik, Technische Universitat Munchen, 1983. — 157 p.
5. **Bauer F.L., Broy M., et.al.** Wide Spectrum Language for Program Specification and Development (Tentative Version). — Munchen, 1981. — 236 p. — (Prepr. / Inst. For Informatik, Technische Universitat Munchen; TUM-18104)
6. **Марков А.А., Нагорный Н.М.** Теория алгорифмов. — М.: Наука, Физматлит, 1984. — 432с.
7. **Минский М.** Вычисления и автоматы / Пер. с англ. — М.: Мир, 1971. — 364 с.
8. **Котов В.Е., Сабельфельд В.К.** Теория схем программ. — Новосибирск: Наука, 1992.
9. **Питерсон Дж.** Теория сетей Петри и моделирование систем. — М.: Мир, 1984.

10. **Мир** Лиспа / Том 2. Методы и системы программирования / Пер. с англ. — М.: Мир, 1990. —318 с.
11. **Gilmore S.** Programming in Standard ML ’97: A Tutorial Introduction. — 1997. — 68 p. — (Prepr. / University of Edinburg; ESC-LFCS-97-364)
12. **Бирюкова Ю.В.** Sisal 90. Руководство пользователя. — Новосибирск, 2000. — 83 с. — (Препр. / СО РАН. ИСИ; № 72.)
13. **Вирт Н.** Программирование на языке Модула-2 / Пер.с англ. — М.: Мир, 1987. — 222 с.
14. **Armstrong J.L., Virding S.R., Williams M.C.** Erlang. User’s Guide & Reference Manual. Version 3.2. — Ellemtel Utvecklings AB, 1991. — 45p.
15. **Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadek F.K.** Efficiently computing static single assignment form and the control dependence graph // ACM Trans. on Programming Languages and Systems. — 1991. — Vol. 13, N 4. — P. 451–490.
16. **Jay C.B., Stecker P.A.** The Functional Imperative: Shape! // Programming languages and systems: ESOP'98. — 1998. — P. 139–153. — (Lect. Notes Comput. Sci.; 1381).
17. **Jay C.B.** Partial evaluation of shaped programs: experience with **FISh**. — University of Technology, Sydney, 1999. — 12 p.
18. **Abdallah A.E.** Derivation of parallel algorithms from functional specifications to CSP processes // Mathematics of Programm Construction. — 1995. — P. 67–96. — (Lect. Notes Comput. Sci.; 947).
19. **Hoare C.A.R.** Communicating Sequential Processes. — Prentice-Hall, 1985.
20. **Achatz K., Schulte W.** Architecture independent massive parallelization of Divide-and-Conquer algorithms. // Mathematics of Programm Construction. — 1995. — P. 97–128. — (Lect. Notes Comput. Sci.; 947).
21. **Wilde D.** The ALPHA language. — Rennes, 1994. — 23 p. — (Rapp./INRIA; No.2295).
22. **Quinton P., Rajopadhye S., Wilde D.** Using static analysis to derive imperative code from ALPHA. — Rennes, 1994. — 20 p. – (Rapp./INRIA; No.2286).
23. **Wilde D., Sie O.** Regular array synthesis using ALPHA. — Rennes, 1994. — 15 p. — (Rapp./INRIA; No. 2289).

В.И.Шелехов

ВВЕДЕНИЕ В ПРЕДИКАТНОЕ ПРОГРАММИРОВАНИЕ

Препринт
100

Рукопись поступила в редакцию 14.06.02
Рецензент Н. В. Шилов
Редактор З. В. Скок

Подписано в печать 04.09.02
Формат бумаги 60 × 84 1/16 Объем 4.7 уч.-изд.л., 5.2 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6