ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет информационных технологий

В. И. ШЕЛЕХОВ

Предикатное программирование

Учебное пособие

Новосибирск
2009

УДК 004.432.42
ББК 22.183.492
Ш 427

Шелехов В. И. Предикатное программирование: Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2009. 109 с.

ISBN 978-5-94356-878-7

Предназначено для студентов факультета информационных технологий и механико-математического факультета, а также всех желающих изучить формальные методы корректности программ и построения эффективных программ в парадигме предикатного программирования. Пособие включает содержание первого семестра курса «Предикатного программирование».

Издание подготовлено в рамках реализации *Программы развития государственного образовательного учреждения высшего профессионального образования «Новосибирский государственный университет»* на 2009–2018 годы.

ISBN 978-5-94356-878-7

Оглавление

Введение в курс предикатного программирования

В представленном учебнике изложен материал лекций по предикатному программированию, которые читались в Новосибирском государственном университете с 2006 по 2009 гг. Данный курс имел два названия: «Формальные методы в описании языков и систем программирования» для студентов 2-го курса магистратуры факультета информационных технологий и «Предикатное программирование» для студентов механико-математического факультета.

Курс начинается общеметодологическим определением понятия программы с рассмотрением двух основных свойств программы: автоматической вычислимости и наличия спецификации. Во втором разделе излагается модель корректности программ для функциональных и императивных программ. Определяется понятие спецификации программы в форме тройки Хоара. Вводится понятие логической семантики. Определяются две системы правил доказательства корректности для оператора суперпозиции, параллельного оператора и условного оператора. В третьем разделе излагаются отдельные положения математической логики: отношения порядка, наименьшая неподвижная точка и метод полной структурной математической индукции. Четвертый раздел описывает язык исчисления вычислимых предикатов CCP (Calculus of Computable Predicates) и его формальную (логическую и операционную) семантику. Для всех конструкций языка и программы в целом доказывается согласованность логической и операционной семантики. Рекурсивные определения предикатов трактуются с использованием аппарата наименьшей неподвижной точки. Язык CCP определяется как минимальное ядро, из которого можно построить любой чистый язык функционального программирования; при этом семантика языка выражается через семантику конструкций ядра. В пятом разделе определяется расширяющаяся цепочка языков CCP $\subset$ P1 $\subset$ P2 $\subset$ P3. Язык P1 получается из CCP применением открытой подстановки определений предикатов на место их вызовов. В языке P2 определяются обобщенные формы оператора суперпозиции и параллельного оператора. Язык P3 определяет набор конструкций, которые в языках императивного программирования соответствуют понятию выражения. Для языков P2 и P3 определяется система правил доказательства корректности новых языковых конструкций. Правила доказательства корректности рекурсивных предикатов определяются в виде правил структурной полной математической индукции.

В шестом разделе в полном объеме описывается последняя версия языка предикатного программирования, оформленная в стиле серии языков C, C++ и др., привычном для подавляющего большинства программистов. В седьмом разделе описываются технология предикатного программирования и методы доказательства корректности предикатных программ. Представлен метод управляемой полуавтоматической трансляции применением последовательности трансформаций, преобразующих предикатную программу в эффективную императивную программу на императивном расширении языка P. Используются трансформации четырех видов: склеивание переменных, реализующее замену нескольких переменных одной; замена хвостовой рекурсии циклом; подстановка определения предиката на место его вызова; кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей. Описывается универсальный метод обобщения исходной задачи для получения решения с хвостовой формой рекурсии. Этот метод нередко оказывается ключевым для эффективной реализации предикатных программ. Предлагается языковая конструкция нового вида – гиперфункция, позволяющая гибко декомпозировать программу произвольным образом.

1. Общее понятие программы

Понятие программы обычно определяется в контексте некоторого языка программирования. Представление об общем понятии программы, абстрагированное от конкретного языка программирования, для специалиста в области информатики суммируется из его опыта работы с разными языками программирования. В литературе хорошо представлены различные аспекты языков программирования: типы данных, модульность, объектная ориентированность, параллелизм и др. (см., например, [13]). Однако сумма разных аспектов не дает четкой интегрированной концепции общего понятия программы. Этот пробел обнаруживается в учебных курсах по основам программирования и в энциклопедиях. Возможно, лучшим кратким определением в энциклопедиях является следующее: «программа есть описание алгоритма решения задачи, заданное на языке программирования» [1].

Понятия алгоритма и программы не тождественны. Понятие алгоритма известно давно и считается хорошо изученным. В математике существуют следующие формализации понятия алгоритма: машина Тьюринга [10], частично рекурсивные функции, нормальный алгорифм Маркова [2] и каноническая система Поста [3]. Понятие программы появилось вместе с первыми компьютерами в 1940-х [1].

Имеются два определяющих свойства программы: автоматическая вычислимость и существование спецификации.

1.1. Автоматическая вычислимость

Развитие средств вычислений сопровождается все большей степенью автоматизации вычислений. Потребность в проведении сложных и длительных расчетов привела человека к необходимости создания и совершенствования механических устройств и машин для облегчения процесса вычисления. С древних времен известно устройство, которое в России называется «счеты». Счеты можно встретить и в наши дни. Счетные машины и арифмометры, создаваемые в XVII–XIX вв., автоматизировали выполнение четырех арифметических действий. В середине XX столетия актуальные потребности науки и практики привели к созданию новой отрасли техники – конструированию и производству счетно-решающих устройств, т. е. приборов и машин для решения математических задач [11]. Применение счетно-решающих устройств становится массовым: они используются для быстрых расчетов в боевой обстановке, в навигации, в автоматизированном управлении сложными агрегатами и т. д. Однако производство счетно-решающих устройств было сложным и дорогим. Поэтому появление универсальных электронно-вычислительных машин (ЭВМ), умеющих автоматически выполнять программу, построенную для произвольного алгоритма, стало неизбежным.

Промежуточное состояние автоматизации вычислений алгоритма реализуется современными калькуляторами, сравнимыми по своим возможностям со счетно-решающими устройствами 1940-х гг. На очередном шаге своей работы калькулятор либо принимает данные от пользователя, либо в автоматическом режиме вычисляет одну из операций. Более развитые калькуляторы позволяют сохранять промежуточные значения вычислений, именовать отдельные значения и массивы значений и использовать эти имена в дальнейших вычислениях. Таким образом, эти калькуляторы могут хранить в своей памяти значения нескольких переменных. Подчеркнем, что в сравнении с ручным счетом калькулятор автоматизирует вычисление только одной операции на каждом шаге, а между шагами требуются действия человека. С помощью калькулятора можно провести расчеты в принципе по любому алгоритму. Если мы захотим исключить действия человека между шагами и ограничить его действия лишь вводом начальных данных, то потребуется каким-то способом задать всю сово-

[1] Первые программы писала математик Ада Лавлейс, дочь поэта лорда Байрона, для спроектированной, но не реализованной вычислительной машины Чарльза Беббиджа в 1830-х гг.

купность операций алгоритма и указать порядок вычисления операций. Необходимым инструментом для этого является программа.

Программа есть алгоритм, реализованный таким способом и в такой форме, что вычисление алгоритма проводится автоматически. По способу реализации различаются аппаратно реализованные программы (например, в виде интегральной схемы) и программы на языках программирования. Автоматическая вычислимость является неотъемлемым свойством программы.

Первые программы для реализации простейших операций появились в аппаратной реализации как составные блоки счетно-решающих устройств. *Аппаратно реализованная программа* конструировалась в виде электронной схемы с использованием электронно-вакуумных ламп, резисторов и конденсаторов, т. е. на элементной базе для ЭВМ первого поколения. Автоматическое исполнение программы реализуется прохождением электрического сигнала по элементам электронной схемы, что обеспечивает на выходе схемы требуемое вычисление. В настоящее время аппаратно реализованная программа конструируется в форме интегральной микросхемы.

Язык программирования определяет правила представления программы в виде текста в конечном алфавите символов. Описание языка определяет: типы данных, структуру памяти исполняемой программы, виды языковых конструкций программы, правила исполнения конструкции каждого вида и программы в целом. Память исполняемой программы состоит из сегментов. Сегмент памяти содержит набор переменных и соответствует независимо исполняемому фрагменту программы (например, процедуре). Языковая конструкция – независимая часть программы: оператор, выражение, описание процедуры и т. д. Всякая конструкция принадлежит некоторому виду. Для вида определяется единый способ изображения (кодирования) конструкций. Вид имеет несколько позиций, в которые помещаются подконструкции. Например, вид Оператор присваивания, изображаемый композицией: <Переменная> := <Выражение>, имеет две позиции для подконструкций, соответствующих понятиям <Переменная> и <Выражение>. Для большинства видов конструкций определен процесс исполнения, составляемый из процессов исполнения подконструкций и связующих действий. Данная концепция описания языка программирования более детально представлена в работе [8]. Отметим, что приведенное определение не годится для языка логического программирования, где исполнение определяется в форме логического вывода.

Автоматическое исполнение произвольной программы на языке программирования реализуется процессором. *Процессор языка программирования* является либо программой, реализованной аппаратно или на некотором другом языке программирования, либо виртуальным. Аппаратно реализованный процессор входит в состав компьютера и определяет язык системы команд компьютера. Примером реализации процессора на другом языке программирования является программа интерпретатора байткода для языка Java [5]. Процессор языка высокого уровня является виртуальным, представленным лишь в описании языка, и не существует в действительности. Язык реализуется посредством трансляции программы на другой язык, для которого существует реализованный процессор.

1.2. Спецификация программы

Для определения связи программы со своей спецификацией рассмотрим следующую модель применения программы (рис. 1). Вычисление по программе является необходимым звеном в цепочке действий некоторого реального процесса, который будем называть *базовым процессом* программы. Вычисление применяется для преобразования информации. Результат преобразования используется в дальнейших действиях процесса. Взаимодействие программы с процессом реализуется через входные и выходные информационные потоки программы. Части процесса, непосредственно взаимодействующие с вычисляемой программой и реализующие потоки, определяют *окружение* программы.

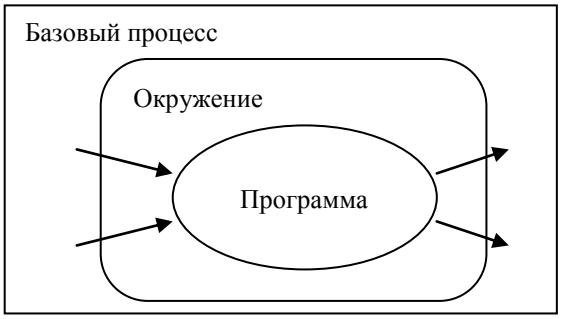

Рис. 1. Модель применения программы

Описание преобразования информации, реализуемого вычислением программы, называется *спецификацией* программы. Спецификация описывает связь между входными и выходными потоками информации. Связь, описываемая спецификацией, имеет объективный характер и отражает реально существующие отношения (зависимости) между объектами базового процесса. Связь либо отражает закон природы, выраженный в математической форме, либо является интерфейсом, спроектированным человеком. В любом случае связь является законом для программы.

В качестве спецификации может выступать любое описание на произвольном языке спецификации. Таким языком часто оказывается язык предметной области, используемый для описания базового процесса.

Проблематика спецификации программ обширна и здесь не рассматривается. Мы ограничиваемся лишь представлением некоторых свойств, отражающих основные связи между спецификацией и программой.

Свойство 1. *Спецификация первична по отношению к программе.* Действительно, спецификация появляется раньше программы. Спецификация необходима человеку для построения программы. Однако для программы, созданных не человеком, спецификация может и не существовать. Примером подобной программы является ДНК, за которой укрепился термин «биокомпьютер».

Свойство 2. *Программа должна соответствовать спецификации.* По определению входные и выходные информационные потоки программы должны соответствовать спецификации. Однако на практике возможны несоответствия между спецификацией и программой, что может иметь следствием неадекватный результат базового процесса, поскольку исполнение программы, как звено базового процесса, не выполнит своей функции.

Свойство 3. *Будучи написанной, программа всегда существует и является строго определенной в силу строгости языка программирования. Спецификация, напротив, может быть нестрогой, неполной, устаревшей, отсутствовать или быть ошибочной.* Этот парадокс типичен для современного программирования. Спецификация существует при разработке программы, однако формы ее представления могут быть самыми различными. Спецификация может быть математически строгой и при этом существовать лишь в голове разработчика программы. При исполнении программы спецификация не нужна. Поэтому модификации программы не всегда сопровождаются соответствующими корректировками спецификации программы.

Спецификация программы не нужна также и при разработке транслятора с языка программирования – здесь программа рассматривается только в качестве теста для транслятора. Разумеется, описание языка программирования никак не касается спецификации. Как следствие, в представлении программиста спецификация часто становится чем-то необязательным. Отметим, что концепция спецификации появилась лишь при разработке больших программных комплексов в системном программировании, т. е. значительно позже первых программ.

Свойство 4. *Программа на языке программирования является абсолютно точной и полной спецификацией самой себя*. Следует из того, что язык программирования с абсолют-

ной строгостью определяет процесс исполнения и результаты для каждого набора входных данных. Из-за отсутствия или неполноты спецификации программа может оказаться единственно существующей спецификацией. Отметим, что программа является неадекватной в качестве спецификации самой себя.

Свойство 5. *Спецификация может быть невычислимой. Программа автоматически вычислима*.

1.3. Формы спецификации программы

Распространена точка зрения, что спецификацию программы можно представить в виде функции. Поскольку это не всегда так, возникает вопрос о классификации форм, которые может принимать спецификация программы. Определим следующие три формы спецификации программы: предикатную, процессную и процессорную.

Спецификация называется *предикатной*, если она эксплицируема в виде формулы на языке исчисления предикатов [2]. Такая экспликация возможна для спецификации, представленной в математической форме. Для программы с предикатной спецификацией характерна простая схема взаимодействия программы с окружением: ввод входных данных происходит в начале исполнения программы, вывод результирующих данных – в конце, а других взаимодействий с окружением нет. В данной схеме спецификация определяет функцию, отображающую значения входных данных в значения результатов.

Предикатная спецификация есть условие математической задачи, исходными данными которой являются входные данные программы, а неизвестными – результаты программы. Программа является реализацией алгоритма решения математической задачи. Алгоритм строится с использованием *свойств* (утверждений, лемм, теорем), доказуемых из условия задачи строгими математическими методами. Описание алгоритма обычно является содержательным, хотя доказательство правильности алгоритма остается математически строгим. Программа является результатом кодирования алгоритма на языке программирования.

Спецификация называется *процессной*, если эффект исполнения программы определяется в виде процесса. Процесс исполнения программы обычно определяется как бесконечный. При исполнении программа обменивается информацией с окружением, причем взаимодействие программы с окружением может быть произвольной сложности. Довольно часто процессная спецификация определяется в виде набора взаимодействующих параллельных процессов. Структура спецификации описывается следующими моделями: машиной конечных состояний [6], сетью Петри [4], моделью CSP [12] Т. Хоара, машиной абстрактных состояний [7] Ю. Гуревича и др. Предикатная спецификация может рассматриваться как частный случай процессной. Однако большинство процессных спецификаций нельзя представить в виде предикатных.

Разработка программы с процессной спецификацией является частью более общей задачи по созданию системы, функционирование которой реализует базовый процесс. В отличие от математических задач спецификация обычно не существует изначально и вырабатывается в процессе конструирования системы. Принято считать, что спецификация является результатом этапа определения требований в разработке системы и программы как ее части.

Спецификация называется *процессорной*, если она описывает функционирование процессора некоторого языка программирования. Спецификация определяет единую форму множеств исполнений произвольной программы на данном языке. Спецификация процессора является частью спецификации языка программирования и определяется семантикой исполнения программы. В пользовательском описании языка программирования спецификация процессора представлена частично и неявно. Формализованное описание семантики ис-

[2] Здесь и далее имеется в виду исчисление предикатов высших порядков.

полнения, абстрагированное от синтаксиса и статической семантики [3], применяется для проектирования внутреннего представления программы в трансляторе [8].

Формальная семантика есть математическое описание семантики языка программирования. Она используется преимущественно для целей верификации программ. Различают следующие виды формальной семантики: операционную, денотационную и аксиоматическую [14; 15]. В большинстве подходов описание формальной семантики оказывается сложным и громоздким. Перспективным является метод алгебраической семантики (разновидности денотационной), в которой состояние исполняемой программы определяется в виде многоосновной алгебры [16; 17], компоненты которой соответствуют один в один состоянию памяти исполняемой программы и операциям по манипулированию ячейками памяти. Описание алгебраической семантики в определенном смысле подобно трансляторной модели программы. Оно прозрачно и компактно, дает возможность однозначного понимания языка и становится полезным для разработчика транслятора.

Известно, что процессор для чистого языка функционального программирования можно написать на том же языке, из чего следует, что спецификация процессора представима в виде функции. Например, интерпретатор Lisp-программы можно написать на языке Lisp [9]. Однако спецификация процессора языка императивного программирования не может быть предикатной. Косвенно это следует из того, что спецификация процессора не может быть проще спецификации программы, исполняемой процессором. По нашей гипотезе процессорная спецификация сложнее процессной, поскольку процессор находится на метауровне по отношению к программе (с процессной спецификацией), которую он исполняет.

Представленные три формы спецификации определяют три принципиально различных класса программ. В первый класс входят программы для математических задач. Второй класс включает программы, работающие в режиме реального времени. Третий класс содержит значительную часть программ системного программирования. Методы разработки программ одного класса не всегда переносимы на программы другого класса.

Список литературы

1. Программа // Краткая российская энциклопедия. М.: Большая российская энциклопедия ОНИКС 21 век, 2003.
2. *Марков А. А., Нагорный Н.М.* Теория алгорифмов. М.: Наука, Физматлит, 1984. – 432 с.
3. *Минский М.* Вычисления и автоматы: Пер. с англ. М.: Мир, 1971. – 364 с.
4. *Питерсон Дж.* Теория сетей Петри и моделирование систем. М.: Мир, 1984. – 264 с.
5. *Gosling J., Joy B., Steele G.* The Java Language Specification. Pre-Release Version 1.0, Draft 5.2 – July 3, 1996.
6. *Kain R. Y.* Automata Theory: Machines and Languages. – McGraw-Hill, N. Y., 1972. – 301 p.
7. *Gurevich Y.* Sequential Abstract State Machines capture Sequential Algorithms // ACM Transactions on Computational Logic. – 2000. – Vol. 1, N 1. – P. 77–111.
8. *Шелехов В. И.* Инвариант языка программирования // Средства и инструменты окружений программирования. – Новосибирск, 1995. – С. 6–22.
9. Мир Лиспа: Пер. с англ. М.: Мир, 1990. Т. 2. Методы и системы программирования. – 318 с.
10. *Клини С. К.* Введение в метаматематику, пер. с англ., М., 1957.
11. *Ершов А .П., Шура-Бура М. Р.* Пути развития программирования в СССР. – Новосибирск, 1976. (Препринт ВЦ СО АН СССР, №12)

[3] К статической семантике относятся правила идентификации переменных и других объектов программы, совокупность ограничений для значений атрибутов конструкций и система умолчаний.

12. *Hoare C. A. R*. Communicating Sequential Processes. Prentice-Hall, 1985.

13. *Watt D.A.* Programming Language Concepts and Paradigms. Prentice Hall, 1990. – 322 p.

14. *Лавров С.С*. Программирование. Математические основы, средства, теория. – СПб: БХВ-Петербург, 3001. – 320 с.

15. *Meyer B.* Introduction to the Theory of Programming Languages. Prentice Hall, 1990. – 448 p.

16. *Zamulin A. V.* Algebraic Semantics of an Imperative Programming Language as a Compiler Abstract Model // Joint NCC\&ISS Bull., 20. – Comp. Science, 2004. – P. 113–128.

17. *Замулин А. В*. Алгебраическая семантика императивного языка программирования // Программирование. 2003, № 6. – С. 1–14.

2. Корректность программ с предикатной спецификацией

Для класса программ с предикатной спецификацией рассматривается программа вместе со своей спецификацией. Введенное в предыдущем разделе понятие предикатной спецификации уточняется следующим образом: спецификация есть формула P(x) & Q(x, y), где P(x) – *предусловие*, истинное перед исполнением программы, а Q(x, y) – *постусловие*, истинное после исполнения программы. Программа S со спецификацией записывается в виде известной тройки Хоара {P(x)} S {Q(x, y)}, см. [9].

Представление о правильности программы принято формулировать в виде понятия *корректности* программы. Корректность определяется *условиями корректности*, которые должны быть истинны для правильной программы. Истинность постусловия после завершения исполнения программы, безусловно, является условием корректности программы. Однако данное условие имеет смысл, если программа завершается. Условие того, что программа завершается, также является условием корректности. Поэтому истинность постусловия принято считать *частичным* условием корректности. Истинность постусловия вместе с условием завершения программы определяет условие *полной* (или *тотальной*) корректности программы.

Чтобы полностью записать условие корректности, определяющее истинность постусловия, следует в математическом виде представить эффект исполнения программы. Необходимым инструментом для этого является формальная семантика языка программирования. Различаются следующие виды формальной семантики: денотационная [8], аксиоматическая [9, 10] и операционная [11]. Мы будем использовать новый вид формальной семантики – *логическую семантику* [2].

Используя логическую семантику, можно построить формулу тотальной корректности программы.

Нашей задачей является разработка методов доказательства корректности программы «вручную», чтобы программист, разработав программу и спецификацию, мог бы доказать ее корректность в традиционном математическом стиле. Здесь необходима техника доказательства снизу вверх: если оператор S состоит из операторов A и B, то сначала следует доказать корректность операторов A и B, а затем по определенным правилам доказывать корректность оператора S. Мы построим систему правил доказательства для трех базисных операторов: оператора суперпозиции A; B, параллельного оператора A || B и условного оператора **if (**C) A **else** B.

2.1. Предикатная спецификация программы

Уточним и дополним понятие предикатной спецификации, определенное в разд. 1.2. Рассмотрим следующую простую схему взаимодействия программы с окружением: ввод входных данных происходит перед началом исполнения программы, вывод результатов – по завершению исполнения, а других взаимодействий с окружением нет. В данной схеме программа реализует функцию, отображающую значения входных данных в значения результатов. Допустим, программа представлена оператором S; x обозначает набор входных данных, а y – набор результатов. Спецификация программы определяется формулой P(x) & Q(x, y), где P(x) – *предусловие*, истинное перед исполнением программы, а Q(x, y) – *постусловие*, истинное после исполнения программы. Программа со спецификацией представляется известной тройкой Хоара {P(x)} S {Q(x, y)}, см. [9].

Спецификация P(x) & Q(x, y) является *предикатной* при условии, что взаимодействие программы с окружением ограничивается вводом данных перед началом исполнения и выводом результатов после завершения. Программа принадлежит классу программ с предикатной спецификацией, если удается переместить ввод всех данных перед началом исполнения

программы, а вывод – после завершения исполнения. Для программы реального времени (реактивной системы) подобный перенос ввода и вывода принципиально невозможен. Такая программа относится к классу программ с *процессной спецификацией* [4; 5], поскольку эффект исполнения программы нельзя представить в виде функции – эффект определяется только в виде процесса. Спецификация может быть адекватно представлена на языке алгебры процессов Р. Милнера [12] или Т. Хоара [13].

2.2. Логическая семантика языка программирования

Всякая программа содержит логику. Это, например, бизнес-логика, извлекаемая нетривиальным анализом из текста программы в процессе реинжиниринга программы [16]. Это также логические формулы, получаемые по программе в соответствии с формальной семантикой языка программирования (денотационной [8] или аксиоматической [9; 10]) в целях формального доказательства свойств программы.

Проанализируем происхождение логики программы. На рис. 2 представлена общая схема, отражающая связи задачи и программы. Программа является реализацией алгоритма решения некоторой математической задачи. Алгоритм строится с использованием *свойств* (утверждений, лемм, теорем), доказуемых из условия задачи. Описание алгоритма обычно является содержательным. Программа является результатом кодирования алгоритма на языке программирования.

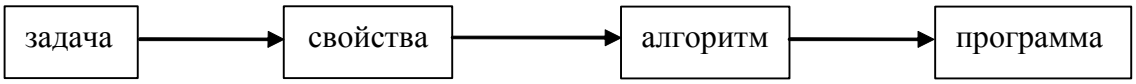

Рис. 2. Связи между задачей и программой

Логика алгоритма включает свойства, использованные для построения алгоритма. Логика программы является трансформацией логики алгоритма вследствие кодирования алгоритма в конструкциях и структурах данных языка программирования. Оптимизации, совершаемые в процессе кодирования алгоритма в императивной программе, обычно приводят к трансформации логики, усложняя (искривляя) ее.

Формальная семантика языка программирования признана элегантным и мощным формализмом, однако не для реально используемого императивного языка программирования. Для него денотационная семантика оказывается сложной и громоздкой из-за сложности самого языка, а аксиоматическая семантика может быть построена лишь для некоторого простого подмножества языка. Эта оценка, данная Джоном Бэкусом 30 лет назад в своей тьюринговской лекции [17], остается справедливой в настоящее время для современных императивных языков. Таким образом, императивные языки – это языки сложной и кривой логики, которую весьма трудно извлечь из программы.

Логическая семантика является простейшим способом определения логики программы. *Логическая семантика* есть функция LS, сопоставляющая каждой исполняемой конструкции S соответствующую формулу $\mathsf{L_S}$ на языке исчисления предикатов, т. е. $\mathsf{LS(S)} = \mathsf{L_S}$.

Определение логической семантики покажем на примере трех операторов языка исчисления вычислимых предикатов ССР [4] [3]. Подробное описание языка ССР будет дано в следующем разделе. Язык ССР определен как минимальное ядро, из которого можно породить любой чистый язык функционального программирования применением иерархической системы обозначений, свойственных языку. Таким способом определен язык предикатного программирования [7]; его описание будет дано в гл. 6.

Пусть A и B обозначают операторы, x, y и z – различные непересекающиеся наборы переменных. Обозначение $\mathsf{A}\colon \mathsf{x} \to \mathsf{y}$ определяет оператор A с набором аргументов x и набором результатов y.

[4] CCP – Calculus of Computable Predicates.

Оператор суперпозиции A; B определяет последовательное исполнение операторов A: $x \to z$ и B: $z \to y$. Логическая семантика оператора суперпозиции определяется следующим образом:

$$LS(A; B)(x, y) \cong \exists z \, (LS(A)(x, z) \,\&\, LS(B)(z, y)) . \quad (2.1)$$

Параллельный оператор A || B определяет параллельное исполнение операторов A: $x \to y$ и B: $x \to z$. Логическая семантика определяется формулой:

$$LS(A \,||\, B)\,(x, y, z) \cong LS(A)(x, y) \,\&\, LS(B)(x, z) . \quad (2.2)$$

Условный оператор **if (**C) A **else** B определяет исполнение одного из операторов A: $x \to y$ и B: $x \to y$ в зависимости от значения логического выражения C, зависящего от x. Логическая семантика определяется формулой

$$LS(\textbf{if (}C) \; A \; \textbf{else} \; B)\,(x, y) \cong (C \Rightarrow LS(A)(x, y)) \,\&\, (\neg C \Rightarrow LS(B)(x, y)) . \quad (2.3)$$

Правильность построения логической семантики определяется сопоставлением с *операционной семантикой* языка программирования. Предикат RUN(S, x, y) обозначает следующее утверждение: существует такое исполнение конструкции S для набора значений x, которое завершается, и результатом исполнения является набор значений y. Данное определение допускает неоднозначность при исполнении S: в результате исполнения может быть получен другой набор y', отличный от y.

Для конструкции S определим *свойство согласованности* (consistency) Cons(S): предикат LS(S) является истинным на наборах значений x и y тогда и только тогда, когда существует исполнение S для набора x, завершающееся вычислением набора значений y. Формально:

$$Cons(S) \cong \forall x \; \forall y \; (LS(S)(x, y) \Leftrightarrow RUN(S, x, y)) . \quad (2.4)$$

Логическая семантика *согласована* с операционной семантикой, если свойство согласованности истинно для всех конструкций языка программирования.

Логическая и операционная семантика языка CCP и доказательство согласованности двух семантик описываются в следующем разделе.

Логическая семантика очевидным образом определяется для языков логического программирования. Однако в этом нет необходимости, поскольку логические формулы, входящие в программу, трактуются в общеизвестном математическом смысле. Построение логической семантики оказывается полезным лишь для нелогических особенностей логического программирования, таких как ситуация «неудачи» при ложном значении в Прологе; описание семантики с учетом данных особенностей дается в четырехзначной логике [14]. Логическая семантика может быть определена для языка предикативного программирования Э. Хехнера [15]. Программа там строится как результат последовательного уточнения (refinement) спецификации программы в виде логической формулы. Уточнения реализуются применением системы правил логического вывода.

Логическая семантика может быть построена для чистых [5] языков функционального программирования. Каждой конструкции языка функционального программирования можно сопоставить эквивалентную формулу на языке исчисления предикатов. Проблематично построение логической семантики для императивных языков. В логической формуле L_S для оператора S кроме переменных, явно встречающихся в операторе S, должны участвовать другие переменные, определяющие состояние памяти исполняемой императивной программы.

[5] Практически все языки функционального программирования содержат расширения, включающие конструкции императивного программирования в целях повышения эффективности программ. Такие языки не являются чистыми.

2.3. Модель корректности программы

Рассмотрим программу с предикатной спецификацией на языке, для которого можно построить логическую семантику. Программа со спецификацией представляется в виде тройки Хоара:

$$\{P(x)\}\ S\ \{Q(x, y)\}\ . \qquad (2.5)$$

Здесь программа представлена оператором S, x – набор входных данных, y – набор результатов.

Закон соответствия программы и спецификации, сформулированный в разд. 2.1, очевидным образом конкретизируется для программы с предикатной спецификацией в виде следующих условий:

- входные данные x должны соответствовать предусловию $P(x)$ перед исполнением программы;
- входные данные x и результаты y должны удовлетворять постусловию $Q(x, y)$ после завершения исполнения программы.

Во втором условии неявно подразумевается, что исполнение программы на входных данных x завершается. Более точная формулировка условия: если для набора x существует исполнение, завершающееся набором y, то истинно постусловие $Q(x, y)$. Используя обозначения, введенные в разд. 2.2, запишем условие в виде формулы: $RUN(S, x, y) \Rightarrow Q(x, y)$. Учитывая свойство согласованности (2.4), второе условие можно представить формулой: $LS(S)(x, y) \Rightarrow Q(x, y)$. В итоге, закон соответствия программы и спецификации для программы с предикатной спецификацией выражается следующей формулой:

$$P(x)\ \&\ (\ LS(S)(x, y) \Rightarrow Q(x, y)\ )\ . \qquad (2.6)$$

Замечание. Импликация в обратную сторону, т. е. $Q(x, y) \Rightarrow LS(S)(x, y)$, в общем случае неверна. Во многих задачах спецификация $Q(x, y)$ представляется более слабым (общим) условием, допускающим для фиксированного набора x не единственное решение y.

Предусловие $P(x)$ не зависит от оператора S, поскольку значение $P(x)$ определено до исполнения оператора S. Для программы в целом предусловие $P(x)$ считается истинным априори. Истинность предусловия для случая, когда оператор S является частью программы, должна быть доказана, см. разд. 2.4 и 2.5. В любом случае, исполнение оператора S не рассматривается для ложного предусловия. Таким образом, истинность предусловия $P(x)$ определяется как необходимое условие для тройки Хоара (2.5).

Закон соответствия программы и спецификации (2.6) определяет условие корректности программы (2.5). Однако первый конъюнкт $P(x)$ не может входить в условие корректности оператора S, а второй конъюнкт реализуется при условии истинности $P(x)$. Учитывая это, условие корректности программы представляется следующей формулой:

$$P(x)\ \&\ LS(S)(x, y) \Rightarrow Q(x, y)\ . \qquad (2.7)$$

Предикат $H(x, y)$ *реализуем* для конкретного x, если $H(x, y)$ истинен для некоторого y, т. е. $\exists y.\ H(x, y)$. Предикат $H(x, y)$ *реализуем* в области предусловия $P(x)$, если он реализуем для всех x, на которых истинно предусловие, т. е. $\forall x.\ (P(x) \Rightarrow \exists y.\ H(x, y))$. Предикат $H(x, y)$ является тотальным в области предусловия, если рассматривать его как функцию, отображающую x в y. Спецификация в виде предусловия $P(x)$ и постусловия $Q(x, y)$ *реализуема*, если постусловие реализуемо в области предусловия.

Рассмотрим ситуацию, когда условие корректности (2.7) истинно, а спецификация нереализуема. Тогда для некоторого x истинно предусловие $P(x)$, а постусловие $Q(x, y)$ ложно для всех y. Из истинности (2.7) следует, что $LS(S)(x, y)$ будет ложной для всех y. Как следствие свойства согласованности (2.4) получим, что $\forall y\ \neg RUN(S, x, y)$ истинно, т. е. исполнение оператора S для данного x не определено. Это означает, что исполнение либо реализуется бесконечно, либо завершается аварийно без получения результата. Подобная ситуация

абсурдна в модели применения программы (см. рис. 1) – такое исполнение бессмысленно и не соответствует цели базового процесса.

Таким образом, мы приходим к следующему условию *корректности спецификации*: спецификация должна быть реализуема, т. е.:

$$P(x) \Rightarrow \exists y.\ Q(x, y)\ . \qquad (2.8)$$

Корректность спецификации является необходимым условием корректности программы, представленной тройкой (2.5).

Определим *точное предусловие* [6] $PE(x)$ следующей формулой:

$$PE(x) \cong \exists y.\ Q(x, y)\ .$$

Лемма 2.1. Спецификация с предусловием $P(x)$ и постусловием $Q(x, y)$ реализуема $\Leftrightarrow$ $P(x) \Rightarrow PE(x)$.

Ситуация, когда для некоторого набора x, удовлетворяющего предусловию $P(x)$, исполнение оператора S реализуется бесконечно или завершается аварийно без получения результата, т. е. $\forall y.\ \neg RUN(S, x, y)$, определена выше как недопустимая для программы с предикатной спецификацией. Следовательно, должно быть истинно обратное утверждение: $\exists y.\ RUN(S, x, y)$, т. е. исполнение программы завершается с получением некоторого результата y. Данное утверждение эквивалентно $\exists y.\ LS(S)(x, y)$, что следует из свойства согласованности (2.4). Таким образом, необходимым условием корректности программы является нормальное завершение исполнения программы для x, удовлетворяющего предусловию $P(x)$. Условие корректности определяется формулой:

$$P(x) \Rightarrow \exists y.\ LS(S)(x, y)\ . \qquad (2.9)$$

Приведенные выше требования объединим в понятии корректности программы. Прежде всего, для программы (2.5) предусловие $P(x)$ всегда истинно. Программа *корректна*, если спецификация реализуема (условие (2.8)), программа нормально завершается в области предусловия (условие (2.9)) и программа соответствует спецификации (условие (2.7)).

Требования, составляющие корректность программы, избыточны.

Лемма 2.2. Если программа соответствует спецификации и нормально завершается в области предусловия, то спецификация реализуема.

Доказательство. Пусть истинно $P(x)$. Необходимо доказать истинность $\exists y.\ Q(x, y)$. Из условия (2.9) следует истинность $\exists y.\ LS(S)(x, y)$. Допустим, истинность этой формулы реализуется для некоторого y_0. Из условия (2.7) получаем истинность $Q(x, y_0)$, а значит – и $\exists y.\ Q(x, y)$. □

Таким образом, корректность программы (2.5) определяется двумя условиями: соответствие спецификации (2.7) и нормального завершения (2.9). Объединяя формулы (2.7) и (2.9), получаем итоговую формулу корректности программы:

$$P(x) \Rightarrow [\ LS(S)(x, y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ LS(S)(x, y)\ . \qquad (2.10)$$

Приведенное определение корректности программы соответствует общеизвестному понятию тотальной (или полной) корректности программы. Оно справедливо для программ с предикатной спецификацией и для языков программирования с логической семантикой. Основой данного определения корректности является формализация закона соответствия программы и спецификации для программ с предикатной спецификацией.

2.4. Система правил доказательства корректности операторов

Понятие корректности, введенное для программы в целом, применимо также к произвольным операторам программы. Допустим, для некоторого оператора S имеется спецификация в виде тройки Хоара $\{P(x)\}\ S\ \{Q(x, y)\}$. В качестве оператора S будем рассматривать оператор суперпозиции $A; B$, параллельный оператор $A \,||\, B$ или условный оператор **if (**C**)** A **else** B, определенные в разд. 2.2. Нашей задачей является конкретизация формулы

[6] Родственным, но не совпадающим является понятие слабейшего предусловия, введенное Э. Дейкстра [1]; E. Hehner использует термин «точное предусловие» для уточнения [15].

корректности (2.10) для каждого из трех операторов. Естественным требованием является корректность операторов A и B. Условия корректности каждого из трех операторов формулируются в виде правил вывода, с помощью которых можно доказывать корректность оператора. Правила вывода для этих операторов не используют формулы LS(A) и LS(B) – только спецификации (предусловия и постусловия) операторов A и B встречаются в правилах. Эта особенность является существенным преимуществом правил.

2.4.1. Правила для корректного оператора

Для произвольного корректного оператора B перепишем условия корректности (2.7–2.9) в виде правил вывода.

Лемма 2.3. Пусть имеется оператор B со спецификацией в виде тройки Хоара $\{P_B(x)\}\ B\ \{Q_B(x, y)\}$. Предполагается, что оператор B является корректным, т. е. для него истинна соответствующая общая формула корректности (2.10) и формулы (2.7–2.9). Тогда истинны следующие правила вывода.

Правило E1. $P_B(x) \vdash \exists y.\ Q_B(x, y)$.

Правило E2. $P_B(x) \vdash \exists y.\ LS(B)(x, y)$.

Правило E3. $P_B(x) \vdash LS(B)(x, y) \Rightarrow Q_B(x, y)$.

2.4.2. Правила корректности для параллельного оператора

Рассмотрим спецификацию параллельного оператора в виде тройки Хоара:

$$\{P(x)\}\ A\ ||\ B\ \{Q(x, y, z)\}\ .\qquad (2.11)$$

Допустим, операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, y)\},\qquad \{P_B(x)\}\ B\ \{Q_B(x, z)\}\ .$$

Определим правила, гарантирующие корректность параллельного оператора.

Правило RP1. $P(x) \vdash P_A(x)\ \&\ P_B(x)$.

Правило RP2. $Q_A(x, y), Q_B(x, z) \vdash Q(x, y, z)$.

Лемма 2.4. Пусть предусловие P(x) истинно. Допустим, операторы A и B корректны. Если правила RP1 и RP2 истинны (т. е. правая часть доказуема из левой части для каждого правила), то параллельный оператор (2.11) является корректным.

Доказательство. Для доказательства корректности оператора (2.11) в соответствии с формулой (2.10) достаточно доказать реализуемость LS(A || B)(x, y, z) и выводимость постусловия Q(x, y, z) из LS(A || B)(x, y, z). В соответствии с (2.2) формула LS(A || B)(x, y, z) эквивалентна формуле $LS(A)(x, y)\ \&\ LS(B)(x, z)$.

Из истинности предусловия P(x) по правилу RP1 следует истинность $P_A(x)$ и $P_B(x)$. Далее, по правилу E2 становятся истинными формулы $\exists y.\ LS(A)(x, y)$ и $\exists z.\ LS(B)(x, z)$. Их конъюнкция определяет реализуемость LS(A || B)(x, y, z).

Докажем выводимость постусловия Q(x, y, z) из $LS(A)(x, y)\ \&\ LS(B)(x, z)$. Допустим, истинна формула $LS(A)(x, y)\ \&\ LS(B)(x, z)$. Применим правило E3 для $P_A(x)$ и $P_B(x)$, истинность которых определена выше. Получаем истинность формул $LS(A)(x, y) \Rightarrow Q_A(x, y)$ и $LS(B)(x, z) \Rightarrow Q_B(x, z)$. Как следствие, будут истинны $Q_A(x, y)$ и $Q_B(x, z)$. Применяя правило RP2, получаем истинность постусловия Q(x, y, z). □

2.4.3. Правила корректности для оператора суперпозиции

Рассмотрим спецификацию оператора суперпозиции в виде тройки Хоара:

$$\{P(x)\}\ A; B\ \{Q(x, y)\}\ .\qquad (2.12)$$

Предположим, что операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, z)\},\qquad \{P_B(z)\}\ B\ \{Q_B(z, y)\}\ .$$

Определим правила, гарантирующие корректность оператора суперпозиции (2.12).

Правило RS1. $P(x) \vdash P_A(x) \;\&\; \forall z\,(Q_A(x, z) \Rightarrow P_B(z))$.

Правило RS2. $P(x) \;\&\; \exists z\,(Q_A(x, z) \;\&\; Q_B(z, y)) \vdash Q(x, y)$.

Лемма 2.5. Пусть предусловие $P(x)$ истинно. Допустим, операторы A и B корректны. Если правила RS1 и RS2 истинны, то оператор суперпозиции (2.12) является корректным.

Доказательство. В соответствии с формулой (2.10) достаточно доказать реализуемость $LS(A; B)(x, y)$ и выводимость постусловия $Q(x, y)$ из $LS(A; B)(x, y)$. В соответствии с (2.1) формула $LS(A; B)(x, y)$ определена как $\exists z.(LS(A)(x, z) \;\&\; LS(B)(z, y))$.

Из истинности предусловия $P(x)$ по правилу RS1 следует истинность формул $P_A(x)$ и $\forall z\,(Q_A(x, z) \Rightarrow P_B(z))$. Из истинности $P_A(x)$ и правила E2 следует истинность формулы $\exists z.\, LS(A)(x, z)$. Допустим, для некоторого z_0 формула $LS(A)(x, z_0)$ истинна. Из истинности $P_A(x)$ и правила E3 следует истинность $LS(A)(x, z_0) \Rightarrow Q_A(x, z_0)$. Как следствие, истинно $Q_A(x, z_0)$. Далее, из истинности формулы $\forall z\,(Q_A(x, z) \Rightarrow P_B(z))$ следует истинность $P_B(z_0)$. По правилу E2 истинна формула $\exists y\, LS(B)(z_0, y)$. Далее, истинна конъюнкция $LS(A)(x, z_0) \;\&\; \exists y\, LS(B)(z_0, y)$, и затем – формула $\exists y.\, \exists z.\, (LS(A)(x, z) \;\&\; LS(B)(z, y))$, т. е. доказана реализуемость $LS(A; B)(x, y)$.

Докажем выводимость постусловия $Q(x, y)$ из $LS(A; B)(x, y)$. Пусть $LS(A; B)(x, y)$ истинно, т. е. истинна формула $\exists z.(LS(A)(x, z) \;\&\; LS(B)(z, y))$. Пусть формула истинна для некоторого z_1. По правилу E3 истинна формула $LS(A)(x, z_1) \Rightarrow Q_A(x, z_1)$ и далее – $Q_A(x, z_1)$. Истинность $Q_A(x, z_1)$ и $\forall z\,(Q_A(x, z) \Rightarrow P_B(z))$ влечет истинность $P_B(z_1)$. По правилу E3 истинно $LS(B)(z_1, y) \Rightarrow Q_B(z_1, y)$. Поскольку $LS(B)(z_1, y)$ истинно, то истинно $Q_B(z_1, y)$. Таким образом, истинна правая часть правила RS2, а значит – и левая, т. е. истинно постусловие $Q(x, y)$. □

2.4.4. Правила корректности для условного оператора

Рассмотрим спецификацию условного оператора в виде тройки Хоара:

$$\{P(x)\}\ \textbf{if (}C\text{) } A\ \textbf{else}\ B\ \{Q(x, y)\}\ . \qquad (2.13)$$

Предположим, что операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, y)\}\ , \qquad \{P_B(x)\}\ B\ \{Q_B(x, y)\}\ .$$

Определим правила, гарантирующие корректность условного оператора (2.13).

Правило RC1. $P(x) \;\&\; C \vdash P_A(x)$.

Правило RC2. $P(x) \;\&\; \neg C \vdash P_B(x)$.

Правило RC3. $P(x) \;\&\; C \;\&\; Q_A(x, y) \vdash Q(x, y)$.

Правило RC4. $P(x) \;\&\; \neg C \;\&\; Q_B(x, y) \vdash Q(x, y)$.

Отметим, что условие C зависит от x.

Лемма 2.6. Пусть предусловие $P(x)$ истинно. Допустим, операторы A и B корректны. Если правила RC1, RC2, RC3 и RC4 истинны, то условный оператор (2.13) является корректным.

Доказательство. Для оператора (2.13) необходимо доказать формулу (2.10). В ней дважды встречается подформула $LS(\textbf{if (}C\text{) } A\ \textbf{else}\ B)(x, y)$, определяемая согласно (3) в виде:

$$(C \Rightarrow LS(A)(x, y)) \;\&\; (\neg C \Rightarrow LS(B)(x, y))\ . \qquad (2.14)$$

В соответствии с формулой (2.10) достаточно доказать реализуемость формулы (2.14) и выводимость постусловия $Q(x, y)$ из (2.14).

Допустим, что условие C истинно. Из истинности предусловия $P(x)$ по правилу RC1 следует истинность $P_A(x)$. По правилу E2 истинна формула $\exists y.\, LS(A)(x, y)$. Далее будет истинной формула $\exists y.\, (C \Rightarrow LS(A)(x, y))$. Из истинности C следует истинность формулы $\neg C \Rightarrow LS(B)(x, y)$ и, следовательно, формулы $\exists y.\, [(C \Rightarrow LS(A)(x, y)) \;\&\; (\neg C \Rightarrow LS(B)(x, y))]$. Это обеспечивает реализуемость формулы (2.14) в случае истинности C. Реализуемость (2.14) в случае ложности C доказывается аналогичным образом.

Докажем выводимость постусловия $Q(x, y)$ из формулы (2.14). Допустим, истинна формула (2.14). Пусть C истинно. Тогда истинно $LS(A)(x, y)$. По правилу RC1 истинно $P_A(x)$. По правилу E3 истинна формула $LS(A)(x, y) \Rightarrow Q_A(x, y)$, а значит – и $Q_A(x, y)$. Таким образом, истинна правая часть правила RC3. Тогда истинна левая часть правила, т. е. истинно постусловие $Q(x, y)$. Доказательство истинности постусловия $Q(x, y)$ для случая, когда C ложно, проводится аналогично с использованием правила RC4. □

2.5. Система правил вывода программы из спецификации

Понятие корректности программы, введенное в разд. 2.3, состоит из трех условий (2.7–2.9). Главным из них является условие (2.7) соответствия программы и спецификации, согласно которому постусловие оператора S должно быть доказано из $LS(S)$. Однако в практике построения программ из математического решения задачи, как правило, реализуется вывод в обратную сторону: программа выводится из спецификации. Нашей целью является определения условий, при которых такой вывод был бы возможен, с построением соответствующих правил доказательства корректности программы. Необходимым условием является однозначность спецификации.

2.5.1. Однозначность предикатов

Предикат $H(x, y)$ является *однозначным* в области X для набора переменных x, если он определяет функцию, отображающую значения набора x в значения набора y. Должно быть истинным следующее условие:

$$\forall x \in X \; \forall y_1, y_2 .\; H(x, y_1) \;\&\; H(x, y_2) \Rightarrow y_1 = y_2 \; .$$

Отметим, что однозначный предикат в некоторой области X не обязательно является реализуемым в этой области.

Оператор S является *однозначным* в области X, если формула $LS(S)(x, y)$ однозначна в области X. Нетрудно убедиться, что оператор суперпозиции $A; B$, параллельный оператор $A \,||\, B$ и условный оператор **if (**C) A **else** B являются однозначными, если однозначными являются операторы A и B.

Спецификация оператора S, представленная предусловием $P(x)$ и постусловием $Q(x, y)$, является *однозначной*, если постусловие является однозначным в области предусловия, т. е.

$$\forall x .\; P(x) \Rightarrow \forall y_1, y_2 .\; (Q(x, y_1) \;\&\; Q(x, y_2) \Rightarrow y_1 = y_2) \; .$$

Отметим, что однозначная спецификация не обязательно является реализуемой.

Лемма 2.7. Допустим, предикат $D(x, y)$ является однозначным в области X, а предикат $Z(x, y)$ реализуем в области X. Пусть истинна формула $Z(x, y) \Rightarrow D(x, y)$. Тогда истинна следующая формула: $\forall x \in X .\; D(x, y) \Rightarrow Z(x, y)$. Как следствие, предикаты D и Z оказываются тождественными в области X.

Доказательство. Пусть истинно $D(x, y)$ для некоторого $x \in X$. Докажем истинность $Z(x, y)$. Поскольку предикат Z реализуем, то существует некоторый y', для которого истинно $Z(x, y')$. Из $Z(x, y) \Rightarrow D(x, y)$ получаем истинность $D(x, y')$. Поскольку истинны $D(x, y)$ и $D(x, y')$, то из однозначности D следует $y = y'$. Тогда истинно $Z(x, y)$. □

2.5.2. Теорема тождества спецификации и программы

Теорема определяет условия, при которых программа может быть выведена из спецификации.

Теорема 2.1 тождества спецификации и программы. Рассмотрим программу со спецификацией в виде тройки Хоара:

$$\{P(x)\}\ S\ \{Q(x, y)\}\ ^{7}\ . \qquad (2.15)$$

Допустим, оператор S является однозначным в области истинности предусловия $P(x)$, а спецификация оператора S является реализуемой. Предположим, $LS(S)(x, y)$ выводима из спецификации, т. е.

$$P(x)\ \&\ Q(x, y) \Rightarrow LS(S)(x, y)\ . \qquad (2.16)$$

Тогда программа (2.15) является корректной.

Доказательство. Для доказательства корректности программы (2.15) достаточно доказать истинность формулы (2.10), т. е.

$$P(x) \Rightarrow [\ LS(S)(x, y)\ \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ LS(S)(x, y)\ .$$

Допустим, предусловие $P(x)$ истинно.

Докажем истинность $\exists y.\ LS(S)(x, y)$. Поскольку спецификация предиката S реализуема, то истинна формула $\exists y.\ Q(x, y)$. Пусть эта формула истинна для некоторого y'. Тогда из (2.16) истинна $LS(S)(x, y')$ и, следовательно, $\exists y.\ LS(S)(x, y)$.

Докажем истинность формулы $LS(S)(x, y) \Rightarrow Q(x, y)$. Поскольку истинны $P(x)$ и формула (2.16), то истинна формула $Q(x, y) \Rightarrow LS(S)(x, y)$. В соответствии с леммой 2.7 истинна формула $LS(S)(x, y) \Rightarrow Q(x, y)$, поскольку постусловие $Q(x, y)$ реализуемо, а $LS(S)(x, y)$ – однозначно. □

Лемма 2.8. В условиях теоремы 2.1 истинна следующая формула:

$$P(x) \Rightarrow (LS(S)(x, y) \equiv Q(x, y))\ .$$

Доказательство. Допустим, предусловие $P(x)$ истинно. Истинность формулы $LS(S)(x, y) \Rightarrow Q(x, y)$ установлена при доказательстве теоремы 2.1. Обратная импликация следует из (2.16). □

Как следствие леммы 2.8 спецификация в виде предусловия $P(x)$ и постусловия $Q(x, y)$ оказывается однозначной.

Лемма 2.9. Допустим, программа (2.15) является корректной, а ее спецификация – однозначной. Тогда истинна формула (2.16), т. е. $LS(S)(x, y)$ выводима из спецификации.

Доказательство. Допустим, истинны $P(x)$ и $Q(x, y)$. Докажем истинность $LS(S)(x, y)$. Из корректности программы (2.15) по формуле (2.10) следует истинность формулы $LS(S)(x, y) \Rightarrow Q(x, y)$, а также реализуемость $LS(S)(x, y)$. Пусть для некоторого y' истинно $LS(S)(x, y')$. Тогда истинно $Q(x, y')$, а из однозначности Q следует $y = y'$. Следовательно, истинно $LS(S)(x, y)$. □

Теорема 2.1, сформулированная для программы в целом, применима также к произвольным операторам программы. Предлагаемая ниже система правил доказательства корректности операторов базируется на теореме 2.1, в соответствии с которой для реализуемой спецификации и при условии однозначности используемых операторов требуется доказать истинность формулы (2.16). Отметим, что однозначности спецификации не требуется. Однако в случае неоднозначной спецификации не удастся доказать истинность формулы (2.16).

2.5.3. Правила корректности для параллельного оператора

Рассмотрим спецификацию параллельного оператора в виде тройки Хоара:

$$\{P(x)\}\ A\ ||\ B\ \{Q(x, y, z)\}\ ^{8}\ . \qquad (2.17)$$

Допустим, операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, y)\}\ , \qquad \{P_B(x)\}\ B\ \{Q_B(x, z)\}\ .$$

Определим правила, гарантирующие корректность параллельного оператора.

---

[7] Данная тройка совпадает с (2.5).

[8] Данная тройка совпадает с (2.11).

Правило LP1. $P(x) \vdash P_A(x)\ \&\ P_B(x)$.

Правило LP2. $P(x)\ \&\ Q(x, y, z) \vdash Q_A(x, y)$.

Правило LP3. $P(x)\ \&\ Q(x, y, z) \vdash Q_B(x, z)$.

Лемма 2.10. Допустим, спецификация параллельного оператора (2.17) реализуема, операторы A и B однозначны в области предусловий и корректны, а их спецификации – однозначны. Если правила LP1, LP2 и LP3 истинны (т. е. правая часть доказуема из левой части для каждого правила), то параллельный оператор (2.17) является корректным.

Доказательство. Поскольку операторы A и B однозначны, то и параллельный оператор (2.17) является однозначным. Поскольку спецификация оператора (2.17) реализуема, в соответствии с теоремой 2.1 для доказательства леммы достаточно доказать истинность формулы:

$$P(x)\ \&\ Q(x, y, z) \Rightarrow LS(A \parallel B)(x, y, z)\ .$$

В соответствии с (2.2) формула $LS(A \parallel B)(x, y, z)$ эквивалентна $LS(A)(x, y)\ \&\ LS(B)(x, z)$.

Пусть истинны $P(x)$ и $Q(x, y, z)$. Докажем истинность $LS(A)(x, y)$ и $LS(B)(x, z)$. Из истинности предусловия $P(x)$ по правилу LP1 следует истинность $P_A(x)$ и $P_B(x)$. По правилам LP2 и LP3 становятся истинными $Q_A(x, y)$ и $Q_B(x, z)$. Для предикатов A и B выполняются условия леммы 2.9. Поэтому истинны формулы:

$$P_A(x)\ \&\ Q_A(x, y) \Rightarrow LS(A)(x, y);$$
$$P_B(x)\ \&\ Q_B(x, z) \Rightarrow LS(B)(x, z).$$

Поскольку посылки этих формул истинны, то истинны $LS(A)(x, y)$ и $LS(B)(x, z)$. □

2.5.4. Правила корректности для оператора суперпозиции

Рассмотрим спецификацию оператора суперпозиции в виде тройки Хоара:

$$\{P(x)\}\ A; B\ \{Q(x, y)\}\ ^{9}\ . \qquad (2.18)$$

Предположим, что операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, z)\}\ , \qquad \{P_B(z)\}\ B\ \{Q_B(z, y)\}\ .$$

Определим правила, гарантирующие корректность оператора суперпозиции (2.18).

Правило LS1. $P(x) \vdash P_A(x)$.

Правило LS2. $P(x)\ \&\ Q(x, y)\ \&\ Q_A(x, z) \vdash P_B(z)\ \&\ Q_B(z, y)$.

Лемма 2.11. Допустим, спецификация оператора суперпозиции (2.18) реализуема, операторы A и B однозначны в области предусловий и корректны, а их спецификация B однозначна. Если правила LS1 и LS2 истинны, то оператор суперпозиции (2.18) является корректным.

Доказательство. Поскольку операторы A и B однозначны, то и оператор суперпозиции (2.18) является однозначным. В соответствии с теоремой 2.1 для доказательства леммы достаточно доказать истинность формулы:

$$P(x)\ \&\ Q(x, y) \Rightarrow LS(A; B)(x, y)\ .$$

В соответствии с (2.1) формула $LS(A; B)(x, y)$ эквивалентна $\exists z.(LS(A)(x, z)\ \&\ LS(B)(z, y))$.

Пусть истинны $P(x)$ и $Q(x, y)$. Докажем истинность $\exists z.(LS(A)(x, z)\ \&\ LS(B)(z, y))$. Из истинности предусловия $P(x)$ по правилу LS1 следует истинность $P_A(x)$. Из корректности оператора A следует истинность формул $\exists z.\ LS(A)(x, z)$ и $LS(A)(x, z) \Rightarrow Q_A(x, z)$. Допустим для некоторого z_0 истинно $LS(A)(x, z_0)$. Тогда истинно $Q_A(x, z_0)$. В соответствии с правилом LS2 истинна формула $P_B(z_0)\ \&\ Q_B(z_0, y)$. В соответствии с леммой 2.9 истинна формула

$$P_B(z)\ \&\ Q_B(z, y) \Rightarrow LS(B)(z, y)\ .$$

Тогда истинна $LS(B)(z_0, y)$. В итоге, будет истинна формула $\exists z.(LS(A)(x, z)\ \&\ LS(B)(z, y))$. □

[9] Эта тройка совпадает с (2.12).

2.5.5. Правила корректности для условного оператора

Рассмотрим спецификацию условного оператора в виде тройки Хоара:

$$\{P(x)\}\ \textbf{if (}C\text{) } A\ \textbf{else}\ B\ \{Q(x, y)\}\ ^{10}\ . \qquad (2.19)$$

Предположим, что операторы A и B корректны. Их спецификации представлены тройками:

$$\{P_A(x)\}\ A\ \{Q_A(x, y)\}\ , \qquad \{P_B(x)\}\ B\ \{Q_B(x, y)\}\ .$$

Определим правила, гарантирующие корректность условного оператора (2.19).

Правило LC1. $P(x)\ \&\ Q(x, y)\ \&\ C\ \vdash P_A(x)\ \&\ Q_A(x, y)$.

Правило LC2. $P(x)\ \&\ Q(x, y)\ \&\ \neg C\ \vdash P_B(x)\ \&\ Q_B(x, y)$.

Лемма 2.12. Допустим, спецификация условного оператора (2.19) реализуема, операторы A и B однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LC1 и LC2 истинны, то условный оператор (2.19) является корректным.

Доказательство. Поскольку операторы A и B однозначны, то и условный оператор (2.19) является однозначным. В соответствии с теоремой 2.1 для доказательства леммы достаточно доказать истинность формулы:

$$P(x)\ \&\ Q(x, y) \Rightarrow LS(\textbf{if (}C\text{) } A\ \textbf{else}\ B)(x, y)\ .$$

В соответствии с (2.3) формула $LS(\textbf{if (}C\text{) } A\ \textbf{else}\ B)(x, y)$ эквивалентна

$$(C \Rightarrow LS(A)(x, y))\ \&\ (\neg C \Rightarrow LS(B)(x, y))\ .$$

Пусть истинны P(x) и Q(x, y). Докажем истинность формулы $C \Rightarrow LS(A)(x, y)$. Пусть истинно C. Докажем истинность LS(A)(x, y). Можно применить правило LC1, поскольку истинны P(x), Q(x, y) и C. Получим истинность формулы $P_A(x)\ \&\ Q_A(x, y)$. В соответствии с леммой 2.9 истинна формула

$$P_A(x)\ \&\ Q_A(x, y) \Rightarrow LS(A)(x, y)\ .$$

Поскольку истинна посылка, то истинно LS(A)(x, y). Следовательно, доказана истинность формулы $C \Rightarrow LS(A)(x, y)$. Истинность формулы $\neg C \Rightarrow LS(B)(x, y)$ доказывается аналогично. □

2.6. Заключение

Корректность программы определяется для класса программ с предикатной спецификацией и для языков с логической семантикой. Представленную модель корректности естественно рассматривать как развитие модели Хоара [9]. Использование свойств логической семантики позволяет определить корректность программы в виде явной логической формулы (2.10), в которой программа S представлена логическим эквивалентом LS(S). В случае, когда оператор S является однозначным, а спецификация реализуема и однозначна, можно использовать формулу корректности (2.16) в соответствии с теоремой тождества спецификации и программы.

Важнейшей особенностью всех правил доказательства корректности является то, что они не используют формул LS(A) и LS(B) – только спецификации (предусловия и постусловия) операторов A и B встречаются в правилах. Это определяет универсальную применимость правил для любых языков программирования, в том числе императивных, для которых проблематично построить логическую семантику – для применимости достаточно того, чтобы программная композиция пары операторов A и B по форме соответствовала одному из указанных трех операторов языка ССР.

Системы правил доказательства корректности определяют алгоритм доказательства корректности программ. Данный алгоритм можно использовать для реализации системы автоматической верификации корректности. Эта система будет существенно более эффективной по сравнению с системой верификации, построенной с использованием формул (2.10) и (2.16). Алгоритм доказательства корректности можно также использовать как метод доказа-

[10] Эта тройка совпадает с (2.13).

тельства «вручную», т. е. в традиционном математическом стиле. Этот метод применялся для доказательства корректности нетривиального алгоритма МакКрейта построения дерева суффиксов [6].

Отсутствие операторов цикла в рассматриваемом наборе операторов не позволяет полноценно показать новую технику доказательства корректности на примерах программ. Аналогами циклов в языке ССР являются рекурсивные определения предикатов. Доказательство корректности рекурсивно определяемых предикатов реализуется с применением структурной и полной индукции [2]; см. гл. 3.

Язык ССР может быть использован в качестве ядра при построении любого чистого языка функционального программирования в виде иерархической системы обозначений. Например, рассмотрим обобщенный оператор суперпозиции A; B для операторов вида A: x→z, r и B: z, x→p, где x, z, r и p обозначают разные наборы переменных, причем наборы x и r могут быть пустыми. Обобщенный оператор A; B может быть определен в виде композиции простого оператора суперпозиции и параллельного оператора. При этом правила доказательства корректности для обобщенного оператора суперпозиции нетрудно получить из соответствующих правил для простого оператора суперпозиции и параллельного оператора.

Список литературы

1. *Дейкстра Э.* Дисциплина программирования: Пер. с англ. М.: Мир, 1978. 272 с.
2. *Шелехов В. И.* Модель корректности программ на языке исчисления вычислимых предикатов. Новосибирск, 2007. 51 с. (Препр. / ИСИ СО РАН; № 145).
3. *Шелехов В. И.* Исчисление вычислимых предикатов. Новосибирск, 2007. 24 с. (Препр. / ИСИ СО РАН; № 143).
4. *Шелехов В. И.* Анализ общего понятия программы // Методы предикатного программирования. Новосибирск, 2006. Вып. 2. С. 7–16.
5. *Шелехов В. И.* Язык спецификации процессов // Методы предикатного программирования. Вып.2. Новосибирск, 2006. С. 17–34.
6. *Шелехов В. И.* Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52 с. (Препр. / ИСИ СО РАН; № 115).
7. *Шелехов В. И.* Введение в предикатное программирование. Новосибирск, 2002. 82 с. (Препр. / ИСИ СО РАН; № 100).
8. *Scott D. S. and Strachey C.* Towards a mathematical semantics for computer languages // Computers and Automata. 1971. P. 19–46.
9. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
10. *Floyd R. W.* Assigning meanings to programs // Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. AMS, 1967. P. 19–32.
11. *McCarthy J.* A basis for a mathematical theory of computation // Computer Programming and Formal Systems. 1963. P. 33–70.
12. *Milner R. A.* Calculus of Communicating Systems // Lecture Notes in Computer Science, 1980. Vol. 92.
13. *Hoare C. A. R.* Communicating Sequential Processes. Prentice-Hall. 1985.
14. *Andrews J. H.* A logical semantics for depth-first Prolog with ground negation // Theoretical computer science. 1997. Val. 184. P. 105−143.
15. *Hehner E. C. R.* A Practical Theory of Programming, 2nd edition. 2004; URL: http://www.cs.toronto.edu/~hehner/aPToP/
16. Автоматизированный реинжиниринг программ / Под ред. проф. А. Н. Терехова и А. А. Терехова. СПб.: Изд-во С.-Петерб. ун-та, 2000. 332 с.
17. *Backus J.* Can programming be liberated from the von Neumann style? A. Functional Style and Its Algebra of Programs // Communications of the ACM. 1978. Vol. 21 (8). P. 613−641.

3. Математические основы

В данном разделе собраны известные математические понятия, используемые в дальнейшем изложении.

3.1. Отношения порядка

R называется *бинарным отношением* на множестве D, если $R \subseteq D \times D$. Утверждение $(a, b) \in R$ принято записывать в виде $a\ R\ b$. Для произвольного отношения R используются следующие понятия:

- R *рефлексивно* $\cong \forall a \in D.\ a\ R\ a$,
- R *иррефлексивно* $\cong \forall a \in D.\ \neg\ a\ R\ a$,
- R *симметрично* $\cong a\ R\ b \Rightarrow\ b\ R\ a$,
- R *антисимметрично* $\cong\ a\ R\ b\ \&\ b\ R\ a\ \Rightarrow a = b$,
- R *транзитивно* $\cong\ a\ R\ b\ \&\ b\ R\ c \Rightarrow a\ R\ c$.

Эквивалентностью называется рефлексивное, симметричное и транзитивное бинарное отношение R на D.

Обозначим через $(D, \sqsubseteq)$ непустое множество D с отношением порядка $\sqsubseteq$. $(D, \sqsubseteq)$ – *частично упорядоченное множество* (чум), если $\sqsubseteq$ рефлексивно, антисимметрично и транзитивно; при этом порядок $\sqsubseteq$ называется *частичным*. Частичный порядок $\sqsubseteq$ является *линейным* (или *тотальным*), если $\forall a,b \in D.\ (a \sqsubseteq b \vee b \sqsubseteq a)$. Бинарное отношение $\sqsubset$ называется *строгим частичным порядком*, если $\sqsubset$ иррефлексивно, антисимметрично и транзитивно.

Пусть $(D, \sqsubseteq)$ – частично упорядоченное множество. Для произвольного подмножества $S \subseteq D$ определим следующие понятия. *Наименьшим* (*наибольшим*) элементом S называется $a \in S$, такой что $a \sqsubseteq x$ $(x \sqsubseteq a)$ для всех $x \in S$. Для наименьшего и наибольшего элементов D используются обозначения: $\bot$ (bottom) и $\top$ (top). *Верхней гранью* S называется $a \in D$, такой что $x \sqsubseteq a$ для всех $x \in S$. *Нижней гранью* S называется $a \in D$, такой что $a \sqsubseteq x$ для всех $x \in S$. Наименьшая верхняя грань (least upper bound) подмножества S обозначается $\mathrm{lub}(S)$.

Частично упорядоченное множество $(D, \sqsubseteq)$ называется *нижней* (*верхней*) *полурешёткой*, если для любого его подмножества (в том числе и пустого) существует наименьшая верхняя (наибольшая нижняя) грань. Частично упорядоченное множество называется *полной решёткой*, если оно является верхней и нижней полурешеткой.

Лемма 3.1. Нижняя (верхняя) полурешетка является полной решеткой.

Частично упорядоченное множество $(D, \sqsubset)$ со строгим порядком $\sqsubset$ обладает свойством *обрыва бесконечно убывающих цепей* (well-founded partial order), если всякое непустое подмножество S имеет *минимальный элемент*, т. е.

$$\forall S \subseteq D.\ (S \neq \varnothing \Rightarrow \exists a \in S\ \forall s \in S.\ \neg\ s \sqsubset a)\ . \qquad (3.1)$$

Из данного определения следует, что в подмножестве S не существует бесконечно убывающих цепей.

Множество $(D, \sqsubset)$ с линейным порядком, обладающее свойством обрыва бесконечно убывающих цепей, является *вполне упорядоченным*: любое подмножество содержит нижнюю грань.

Более полное изложение материала по отношениям порядка см. в учебном пособии [1].

3.2. Наименьшая неподвижная точка

Далее будем считать, что $(D, \sqsubseteq, \bot)$ – полная решетка с наименьшим элементом, т. е. $\forall a \in D.\ \bot \sqsubseteq a$.

Пусть $F: D \to D$ – произвольная тотальная (т. е. всюду определённая) функция на D. Функция F называется *монотонной*, если $\forall a,b \in D.\ a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)$. Последовательность $\{a_m\}_{m\geq 0}$ является *возрастающей цепью*, если $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_m \sqsubseteq \ldots$. Для натурального n и $x \in D$ определим: $F_0(x) = x$, $F_{n+1}(x) = F_n(F(x))$.

Лемма 3.2. $\{F_n(\bot)\}_{n\geq 0}$ – для монотонной функции F определяют возрастающую цепь: $\bot \sqsubseteq F(\bot) \sqsubseteq F_2(\bot) \sqsubseteq \ldots \sqsubseteq F_n(\bot) \sqsubseteq \ldots$.

Лемма 3.3. Если F монотонна и $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ – возрастающая цепь, то $F(a_0) \sqsubseteq F(a_1) \sqsubseteq F(a_2) \sqsubseteq \ldots$ – также возрастающая цепь.

Для наименьшей верхней грани цепи будем использовать обозначение: $\cup_{m\geq 0} a_m \cong \mathrm{lub}\ \{a_m\}_{m\geq 0}$. Здесь верхняя грань играет роль предела цепи.

Тотальная функция $F: D \to D$ называется *непрерывной*, если для любой возрастающей цепи $\{a_m\}_{m\geq 0}$ выполняется равенство $F(\cup_{m\geq 0} a_m) = \cup_{m\geq 0} F(a_m)$.

Лемма 3.4. Непрерывная функция является монотонной.

Неподвижной точкой функции F называется решение уравнения $x = F(x)$.

Теорема 3.1 Клини. Пусть F – непрерывная функция. Тогда $\cup_{n\geq 0}\{F_n(\bot)\}$ является наименьшей неподвижной точкой F.

3.3. Математическая индукция

Математическая индукция – это метод доказательства некоторого утверждения $P(n)$ для всех значений натурального параметра n; $n = 0, 1, 2, \ldots$. Доказательство проводится по следующей схеме.

- **Начальный шаг:** утверждение $P(n)$ доказывается для $n = 0$ (*база индукции*);
- **Индуктивный шаг:** утверждение $P(n)$ считается истинным для значения n (*индуктивное предположение*) и доказывается для значения $n + 1$.

Переменная n называется *индукционной переменной*. Метод математической индукции базируется на *аксиоме индукции*:

$$(P(0) \ \&\ \forall k.[\ P(k) \Rightarrow P(k + 1)\ ]\ ) \Rightarrow \forall n.P(n)\ . \qquad (3.2)$$

На практике используются различные обобщения метода в зависимости от природы утверждения $P(n)$. База индукции может быть отлична от нуля. Шаг индукции может быть отрицательным. Индукционных переменных может быть несколько. Эти и другие особенности учитываются методами полной индукции и структурной индукции. Их доказательство реализуется применением классического метода математической индукции. Индукционное предположение для *полной индукции* определяет истинность $P(j)$ для всех $j \leq k$; при его использовании требуется доказать $P(k + 1)$.

Далее будем использовать метод, сочетающий структурную и полную индукцию. Пусть утверждение, которое требуется доказать, есть $W(t)$; $t \in X$. Параметр t определяет одну или несколько индукционных переменных. На множестве X задан строгий частичный порядок $\sqsubset$, удовлетворяющий свойству (3.1) обрыва бесконечных убывающих цепей. Метод структурной (и полной) индукции определяется следующей формулой:

$$\forall t \in X.\ [\ (\forall y \in X.\ y \sqsubset t \Rightarrow W(y)\ ) \Rightarrow W(t)\ ]\ . \qquad (3.3)$$

Если элемент t в формуле (3.3) является минимальным, то формула (3.3) вырождается в $\forall t \in X.\ W(t)$. Это значит, что для минимальных элементов доказательство $W(t)$ надо про-

водить отдельно, что соответствует начальному шагу классической схемы. Пусть истинное значение предиката $\beta(t)$ определяет набор значений t, составляющих базу индукции, которая, по меньшей мере [11], должна содержать все минимальные элементы. Тогда формула (3.3) переписывается в виде

$$\forall t \in X.\ [\ (\beta(t) \Rightarrow W(t)) \ \&\ \forall y \in X.\ (\neg\beta(t) \ \&\ y \sqsubset t \Rightarrow W(y)\ ) \Rightarrow W(t)\ ]\ . \qquad (3.4)$$

Формула (3.4) определяет общую схему доказательства по индукции.

Список литературы

1. *Гончаров С. С*. Математическая логика: Учеб. пособие. Новосибирск, 2007. 166 с.
2. *Hopcroft J. E*., Motwani R., Ullman J. D. Introduction to Automata Theory, Languages, and Computation, 2nd ed. Addison Wesley, 2001. 521 p.
3. *Кнут Д*. Искусство программирования для ЭВМ. М.: Мир, 1976. Т. 1. С. 38–50.
4. *Клини С. К*. Введение в метаматематику: Пер. с англ. М.: 1957. 526 с.
5. Mathematical induction. URL: http://en.wikipedia.org/wiki/Mathematical_induction.

[11] Предикат $\beta(t)$ может оказаться истинным для других элементов t, не являющихся минимальными.

4. Язык исчисления вычислимых предикатов, его логическая и операционная семантика

Нас интересуют языки программирования, допускающие построение логической семантики. Во-первых, следует исключить из рассмотрения языки для реактивных систем, определяющих взаимодействующие параллельные процессы. Адекватной моделью для таких систем является алгебра процессов Р. Милнера [8] или Т. Хоара [9]. Проблематично построение логической семантики для императивных языков. Логическая семантика может быть построена для чистых языков функционального программирования. Каждой конструкции языка функционального программирования может быть сопоставлена эквивалентная формула на языке исчисления предикатов.

Множество вычислимых формул исчисления предикатов будем рассматривать в качестве общего базиса для всех языков, допускающих построение логической семантики. Вычислимые формулы будем записывать на языке *исчисления вычислимых предикатов*. Исчисление определяет набор базисных вычислимых предикатов, три базисных оператора (оператор суперпозиции, параллельный оператор и условный оператор) и механизм рекурсивного определения предикатов. Данный язык исчисления определен в форме языка программирования ССР (Calculus of Computable Predicates). Язык ССР определяет минимальный полный базис, достаточный, чтобы запрограммировать любой алгоритм, реализуемый в классе языков с логической семантикой. Язык неудобен для программирования и не предназначен для этого. Простота языка облегчает изучение математических свойств программ.

Язык ССР определяется в качестве минимального ядра для чистых языков функционального программирования. Система типов данных представлена в виде математической модели. Определяется логическая и операционная семантика языка. Доказывается теорема согласованности логической и операционной семантики.

4.1. Структура программы на языке ССР

Язык ССР (Calculus of Computable Predicates) – язык *исчисления вычислимых предикатов*, определяющий множество вычислимых формул исчисления предикатов. К языку предъявляются два требования: полноты и простоты. Язык ССР должен предоставлять достаточно полный набор конструкций, чтобы запрограммировать любой алгоритм, представимый на некотором чистом функциональном языке. Язык должен быть максимально простым, чтобы облегчить исследование свойств программ.

Программа на языке ССР состоит из набора определений предикатов. Определение предиката $A \equiv K$ сопоставляет вычислимую формулу K определяемому предикату A. Вычислимая формула K представляется в виде параллельного оператора, оператора суперпозиции или условного оператора. Всякий оператор определяет композицию двух предикатов. Вхождение предиката в составе оператора называется *вызовом предиката*. Вызов предиката имеет следующее представление:

$$\varphi(d_1, d_2, ..., d_n: e_1, e_2, ..., e_m) \quad , \tag{4.1}$$

где $n \geq 0$, $m > 0$, φ – имя предиката или переменной предикатного типа (см. разд. 4.2); $d_1, d_2, ..., d_n$ – имена переменных, называемых *аргументами вызова*; $e_1, e_2, ..., e_m$ – имена переменных, различающихся между собой и отличных от $d_1, d_2, ..., d_n$. Переменные $e_1, e_2, ..., e_m$ называются *результатами вызова*. *Вычисление предиката* по значениям аргументов определяет значения результатов. Для предиката (4.1) будем также использовать компактное обозначение $\varphi(d: e)$, где d и e обозначают наборы имен $d_1, d_2, ..., d_n$ и $e_1, e_2, ..., e_m$ соответственно.

В случае, когда требуется вычислить логическое значение предиката φ (т. е. набор e пуст), будем использовать в качестве результата дополнительный параметр b логического

типа и записывать предикат в виде φ(d: b). Предикат φ(d: e) в случае пустого набора d (т. е. без аргументов при n = 0) соответствует константе. Выражений [12] и констант нет в языке ССР.

Определение нового предиката есть конструкция вида

$$A(x: y) \equiv K(x: y) \ . \qquad (4.2)$$

Здесь A обозначает имя определяемого предиката; x и y – наборы переменных, причем все переменные различны; K(x: y) обозначает параллельный оператор, оператор суперпозиции или условный оператор. Переменные набора x называются *аргументами предиката*, а набора y – *результатами предиката*. Определение предиката может быть рекурсивным (см. разд. 4.11).

Произвольный предикат языка ССР либо имеет определение вида (4.2), либо является *базисным вычислимым предикатом*, используемым для константы и элементарной операции. Каждый тип данных однозначно определяется некоторым набором базисных предикатов. Совокупность всех базисных предикатов определяет систему типов данных.

4.2. Система типов данных

Любая переменная характеризуется некоторым типом. Язык ССР является бестиповым в том смысле, что типы переменных не указываются в программе, однако их можно однозначно определить по программе. Система типов описывается далее в виде математической модели.

Тип определяет множество значений. Тип идентифицируется *именем типа*. *Система типов* языка ССР включает примитивные типы, подмножество произвольного типа и структурные типы. Для каждого типа набор операций со значениями типа представлен в виде базисных вычислимых предикатов.

Примитивными типами являются: логический, целый, вещественный и литерный. Для них соответственно будем использовать имена: BOOL, INT, REAL и CHAR. Набор базисных предикатов для примитивных типов соответствует распространенным арифметическим и логическим операциям. Например, базисные предикаты +(x, y: z), –(x, y: z), –(x: y), <(x, y: b) соответствуют операциям z = x + y, z = x – y, y = –x , b = x < y.

Для каждого примитивного типа определены два вида предикатов равенства: =(x: y) и =(x, y: b). Для предиката =(x: y) процессор присваивает переменной y значение переменной x. Для предиката =(x, y: b) процессор определяет значение отношения x = y и присваивает его логической переменной b. Предикат =(d: e) определен также для наборов переменных d и e.

Имеется набор базисных предикатов для задания констант. Предикаты ConsIntZero(: x) и ConsIntOne(: x) определяют целочисленные значения 0 и 1 в качестве значения переменной x.

Далее в определениях типов T, S, X, Y и Z обозначают имена типов.

Новый тип S как *подмножество типа* T вводится определением

$$S = SUBSET(T, x, P, d) \ . \qquad (4.3)$$

Здесь x – переменная типа T, а d – произвольный, возможно пустой, набор переменных, P – имя предиката. Предикат P(x, d: b) является вычислимым, т. е. должен быть определен на языке ССР. Тип S есть множество истинности предиката P(x, d), т. е. $S = \{x \in T \mid P(x, d)\}$. Переменные набора d являются *параметрами* типа S. Примером подмножества типа является тип натуральных чисел:

$$NAT = SUBSET(INT, x, GE0) = \{x \in INT \mid x \geq 0\} \ ,$$

где GE0 есть имя предиката $x \geq 0$. Другим примером является диапазон целых чисел:

[12] Термов в смысле языка исчисления предикатов

$$DIAP(n) = SUBSET(INT, x, IN1\_n, n) = \{x \in INT \mid x \geq 1 \ \& \ x \leq n\},$$

где IN1_n есть имя предиката $x \geq 1 \ \& \ x \leq n$.

Структурный тип определяется в виде композиции других типов, называемых *компонентными* по отношению к структурному. Структурными типами являются: произведение типов (кортеж), объединение типов, множество подмножеств типа и предикатный тип. Базисные предикаты для структурного типа подразделяются на конструкторы, деструкторы и другие операции со значениями типа. *Конструктор* по значениям компонентных типов строит значение структурного типа. *Деструктор* для значения структурного типа определяет соответствующие значения компонент.

Произведение типов

$$Z = X \times Y \tag{4.4}$$

определяет тип Z в виде множества кортежей (x, y), т. е. $Z = \{(x, y) \mid x \in X, y \in Y\}$. Конструктором является предикат ConsStruct(x, y: z), где $x \in X$, $y \in Y$ и $z \in Z$, а деструктором – CompStruct(z: x, y). Для конструктора и деструктора истинно отношение z = (x, y).

Объединение [13] *типов*

$$Z = X + Y \tag{4.5}$$

определяет множество элементов вида (1, x) и (2, y) для типов X и Y, т. е. $Z = \{(1, x) \mid x \in X\} \cup \{(2, y) \mid y \in Y\}$. Первая компонента в этих кортежах (1 или 2) является *тегом* значения. Конструктор ConsUnion1(x: z) строит структурное значение z по некоторому $x \in X$, т. е. z = (1, x). Аналогично, для конструктора ConsUnion2(y: z) имеет место z = (2, y). Деструктор CompUnion(z: i, x, y) для $z \in Z$ определяет, что либо z = (1, x) и i = 1, либо z = (2, y) и i = 2. Если i = 1, то значение y не определено. Если i = 2, то значение x не определено.

Множество подмножеств

$$Z = SET(X) \tag{4.6}$$

для конечного типа X есть $Z = \{z \mid z \subseteq X\}$. Конструктор ConsSetEmpty(: z) определяет пустое множество в качестве значения переменной z. Конструктор ConsSetElem(x : z) определяет z = {x} для $x \in X$. Предикат CompSet(z, x: b) определяет истинность формулы $x \in z$.

Предикатный тип

$$Z = PRED(D: E) \tag{4.7}$$

есть множество вычислимых предикатов вида φ(d: e) для непересекающихся наборов переменных d и e, причем набор d может быть пустым. Типы переменных определяются соответствующими наборами типов D и E. Таким образом, $Z = \{\varphi(d: e) \mid d \in D, e \in E, \varphi \in CCP\}$. Конструктор предиката ConsPred описан в разд. 4.8.

Допустим, набор d не пустой, типы набора D являются конечными, а произвольный предикат $\varphi \in Z$ определен для любого $d \in D$, т. е. $\forall \varphi \in Z \ \forall d \in D \ \exists e \ \varphi(d: e)$. При этих условиях предикат φ(d: e) может трактоваться как *массив*. Переменные набора d называются *индексами* массива, а e – *элементом массива*. Конструктор массива ConsArray описан в разд. 4.9. Деструктор CompArray(φ, d: e) определяет элемент e массива φ для набора индексов d.

Совокупность типов программы представлена типами переменных, встречающихся в программе. Всякий тип либо является примитивным, либо вводится одним из определений вида (4.3)–(4.7).

Другими типами данных в языке ССР являются последовательности и деревья. Эти типы являются производными, а не базисными. Их определение является рекурсивным и использует произведение и объединение компонентных типов. Следует особо подчеркнуть, что типы указателей (pointers) не принадлежат языку ССР. В этом нет необходимости, по-

[13] Размеченное объединение в терминологии Т. Хоара [2].

скольку структуры данных, использующие указатели в программе на чистом функциональном языке, могут быть преобразованы в эквивалентные структуры данных с использованием массивов, последовательностей и деревьев. Использование указателей подразумевается в списках для языков семейств ML и Лисп. Формальная семантика типов данных с указателями достаточно сложна.

Дадим примеры рекурсивных типов. Введем специальный тип: NIL = { **nil** }. Имя **nil** обозначает пустую последовательность. Тип последовательности, составленной из элементов некоторого типа T, вводится следующими определениями:

$$\mathsf{Seq} = \mathsf{NIL} + \mathsf{Seq1};$$
$$\mathsf{Seq1} = \mathsf{T} \times \mathsf{Seq}.$$

Здесь тип T является *параметром* определяемого типа Seq. При использовании типа Seq в правой части определения другого типа в скобках указывается конкретный тип – параметр. Например, строковый тип как последовательность литер имеет определение

$$\mathsf{STRING} = \mathsf{Seq(CHAR)}.$$

Тип списка LIST для языка Лисп представляется следующим набором определений:

$$\mathsf{LIST} = \mathsf{Seq(ATOM)};$$
$$\mathsf{ATOM} = \mathsf{NIL} + \mathsf{NUM} + \mathsf{SYMBOL} + \mathsf{LIST};$$
$$\mathsf{NUM} = \mathsf{INT} + \mathsf{REAL};$$
$$\mathsf{SYMBOL} = \mathsf{STRING}.$$

Определим семантику рекурсивно определяемых типов данных. Рассмотрим следующую систему определений типов:

$$X_k = \varphi_k(X_1, X_2,..., X_n, T_1, T_2,..., T_s);\ k = 1,...,n;\ n > 0;\ s \geq 0\ . \qquad (4.8)$$

Здесь $X_1, X_2,..., X_n$ – имена определяемых типов; φ_k – *типовый терм* вида $Y \times Z$ или $Y + Z$, где в качестве Y и Z могут быть типы $X_1, X_2,..., X_n, T_1, T_2,..., T_s$. Определения типов $T_1, T_2,..., T_s$ не зависят от $X_1, X_2,..., X_n$. Например, система определений для типа LIST включает:

$$\mathsf{LIST} = \mathsf{SeqATOM};$$
$$\mathsf{SeqATOM} = \mathsf{NIL} + \mathsf{Seq1};$$
$$\mathsf{Seq1} = \mathsf{ATOM} \times \mathsf{SeqATOM};$$
$$\mathsf{ATOM} = \mathsf{NIL} + \mathsf{NUM} + \mathsf{SYMBOL} + \mathsf{LIST}.$$

Систему определений рекурсивных типов (4.8) перепишем в векторной форме:

$$X = \varphi(X)\ , \qquad (4.9)$$

где $X = (X_1, X_2,..., X_n)$ – вектор типов и $\varphi = (\varphi_1, \varphi_2,..., \varphi_n)$ – вектор типовых термов. Введем отношение $\sqsubseteq$ на векторах типов:

$$X \sqsubseteq Y \cong \forall k=1..n\ (X_k \subseteq Y_k).$$

Отношение $\subseteq$ определяет нижнюю полурешетку на множестве всех типов с пустым типом $\varnothing$ в качестве минимального элемента. Вектор типов $\Theta = (\varnothing, \varnothing,..., \varnothing)$ является минимальным элементом полурешетки, определяемой отношением $\sqsubseteq$ на множестве векторов типов.

Рассмотрим последовательность векторов типов $\{X^m\}_{m\geq 0}$, определяемую рекуррентно следующим образом:

$$X^0 = \Theta,\ X^{m+1} = \varphi(X^m),\ m \geq 0. \qquad (4.10)$$

Естественно ожидать, что предел последовательности $\{X^m\}_{m\geq 0}$ (если он существует) даст нам неподвижную точку – решение системы (4.9). Рассмотрим построение неподвижной точки для рекурсивного типа Seq. Получим следующую последовательность типов:

$\mathsf{Seq}^0 = \varnothing;$
$\mathsf{Seq}^1 = \mathsf{NIL} + (\mathsf{T} \times \mathsf{Seq}^0) = \mathsf{NIL} + \varnothing = \{1\} \times \mathsf{NIL} \cup \{2\} \times \varnothing = \{(1, \mathbf{nil})\};$
$\mathsf{Seq}^2 = \mathsf{NIL} + \{(t, (1, \mathbf{nil}))\}$, где $t \in T$;
$\mathsf{Seq}^3 = \mathsf{NIL} + (\{t_2\} \times (\mathsf{NIL} + \{(t_1, (1, \mathbf{nil}))\}))$, где $t_1, t_2 \in T$, и т. д.

Лемма 4.1. Функция φ системы (4.9) определений рекурсивных типов $X = \varphi(X)$ является непрерывной относительно произведения и объединения. Понятие непрерывности функций определено в разд. 3.1. Отметим, термы вида SUBSET, SET и PRED могут встречаться в системе (4.9), но они не могут участвовать в рекурсии. Например, недопустима рекурсия вида $Z = SET(Z)$.

Доказательство. Докажем, что произведение $Y \times Z$ и объединение $Y + Z$ непрерывны относительно вхождений компонентных типов Y и Z. Пусть $\{Y^m\}_{m\geq 0}$ и $\{Z^m\}_{m\geq 0}$ – возрастающие цепи типов. Требуется доказать, что

$$\cup_{m\geq 0}(Y^m \times Z^m) = (\cup_{m\geq 0}Y^m) \times (\cup_{m\geq 0}Z^m).$$

Докажем сначала, что тип левой части равенства содержится в типе правой части. Пусть $(y, z) \in \cup_{m\geq 0}(Y^m \times Z^m)$. Существует такое k, что $(y, z) \in Y^m \times Z^m$ для всех $m \geq k$. Тогда $y \in Y^m$ и $z \in Z^m$ для $m \geq k$. Далее, $y \in \cup_{m\geq 0}Y^m$ и $z \in \cup_{m\geq 0}Z^m$. И, как следствие, $(y, z) \in (\cup_{m\geq 0}Y^m) \times (\cup_{m\geq 0}Z^m)$. Допустим теперь, что $(y, z) \in (\cup_{m\geq 0}Y^m) \times (\cup_{m\geq 0}Z^m)$, и докажем, что $(y, z) \in \cup_{m\geq 0}(Y^m \times Z^m)$.

Из определения произведения $y \in \cup_{m\geq 0}Y^m$ и $z \in \cup_{m\geq 0}Z^m$. Существуют такие k_1 и k_2, что $y \in Y^m$ для $m \geq k_1$ и $z \in Z^m$ для $m \geq k_2$. Пусть $k = \max(k_1, k_2)$. Тогда $(y, z) \in Y^m \times Z^m$ для всех $m \geq k$. Далее, $(y, z) \in \cup_{m\geq 0}(Y^m \times Z^m)$. Непрерывность произведения доказана. Непрерывность объединения доказывается аналогичным образом.

Непрерывность функции φ доказывается следующей цепочкой равенств:

$$\cup_{m\geq 0}\varphi(B^m) = \cup_{m\geq 0}(\varphi_1(B^m), ..., \varphi_n(B^m)) = (\cup_{m\geq 0}\varphi_1(B^m), ..., \cup_{m\geq 0}\varphi_n(B^m)) =$$
$$= (\varphi_1(\cup_{m\geq 0}B^m), ..., \varphi_n(\cup_{m\geq 0}B^m)) = \varphi(\cup_{m\geq 0}B^m). \qquad \square$$

В соответствии с теоремой Клини о неподвижной точке (см. разд. 3.1) решением системы $X = \varphi(X)$ является наименьшая неподвижная точка функции φ, т. е. $\cup_{m\geq 0}X^m$, где $X^0 = \Theta$, $X^{m+1} = \varphi(X^m)$.

Для построения последовательностей и деревьев вполне достаточно рекурсии с использованием произведения и объединения. Возможны другие, экзотические, формы рекурсии, однако они бесполезны при разработке реальных алгоритмов. Некоторые из них анализируются в книге [3].

4.3. Логическая и операционная семантика языка ССР

Логическая семантика есть функция LS, сопоставляющая каждой конструкции S языка ССР некоторую формулу L_S в исчислении предикатов, т. е. $LS(S) = L_S$. Пусть $\varphi(d{:}\ e)$ есть вызов предиката (4.1). Тогда

$$LS(\varphi(d{:}\ e)) \cong \varphi(d, e) .$$

Пусть $A(x{:}\ y) \equiv K(x{:}\ y)$ – определение предиката (2). Тогда

$$LS(A(x{:}\ y) \equiv K(x{:}\ y)) \cong A(x, y) \equiv LS(K(x{:}\ y))\ ^{14} .$$

Логическая семантика $LS(K(x{:}\ y))$ для оператора $K(x{:}\ y)$, обозначающего параллельный оператор, оператор суперпозиции или условный оператор, определена в разд. 4.5–4.7. Логическая семантика рекурсивных определений предикатов представлена в разд. 4.11.

Любая конструкция языка ССР легко конвертируется на язык исчисления предикатов. С операторами и предикатами языка ССР можно оперировать как с формулами исчисления предикатов. Поэтому далее будем иногда опускать применение функции LS и говорить об истинности операторов и предикатов.

Операционная семантика определена в виде метапрограммы, реализующей исполнение программы на языке ССР. Метапрограмма записана на *метаязыке* и исполняется *абстрактным процессором*. Ядром метаязыка являются базисные операции с *памятью* исполняемой программы. Память состоит из набора секций. *Секция памяти* содержит конечный набор переменных вместе с их значениями. Секция создается в начале исполнения вызова

[14] Здесь и далее параметром функции LS является произвольная языковая конструкция. Запись функции LS отличается от используемой в гл. 2.

предиката, для которого имеется определение в программе. Секция состоит из аргументов, результатов и локальных переменных определения предиката.

Произвольная секция s представляется массивом. Индексами массива являются имена переменных. Пусть a – имя переменной. Тогда s[a] обозначает значение переменной с именем a в секции s. Если x – набор имен переменных, принадлежащих секции s, то s[x] обозначает соответствующий набор значений. Пусть a – имя переменной, а v – значение того же типа. Тогда s[a] := v обозначает оператор присваивания значения v переменной a в секции s. Будем также использовать групповой оператор присваивания s[x] := w, где x – набор имен переменных; w – набор значений. Оператор s = newSect(A) создает в памяти новую секцию s, соответствующую определению предиката с именем A. Секция s отлична от всех других секций, созданных ранее.

Исполнение вызова предиката φ(d: e) в секции s при условии, что переменные наборов d и e принадлежат секции s, представляется на метаязыке следующей конструкцией:

runCall(s, φ(d: e)) .

Исполнение оператора K(x: y) в секции s записывается на метаязыке следующей конструкцией:

runStat(s, K(x: y)) .

Конструкции метаязыка для исполнения параллельного оператора, оператора суперпозиции или условного оператора определены в разд. 4.5–4.7.

Возможен иной способ определения операционной семантики, предполагающий подстановку правой части определения предиката на место исполняемого вызова предиката. Такой способ используется для функциональных языков программирования (см., например, [1]).

Пусть H(x: y) есть вызов предиката [15] или оператор. Предикат RUN(H, x, y) для фиксированных значений наборов x и y обозначает следующее утверждение: существует такое исполнение H(x: y) для значений набора x, которое завершается, причем результатом являются значения набора y. Следует отметить, что данное определение RUN учитывает случай возможной неоднозначности, если рассматривать H как функцию из x в y. Определим *свойство согласованности* Cons(H):

$$Cons(H) \cong \forall x\ \forall y\ (LS(H(x: y)) \Leftrightarrow \exists y'\ (RUN(H, x, y') \,\&\, eq(y, y'))) . \qquad (4.11)$$

Предикат eq(y, y′) определяет *тождественность* значений наборов y и y′ по следующим правилам. Пусть набор y есть $y_1, \ldots, y_m$, $m>0$, а y′ есть $y'_1, \ldots, y'_m$. Тогда истинны $eq(y_1, y'_1)$, … $eq(y_m, y'_m)$. Тождественность двух значений примитивного типа, а также типа множества подмножеств, определяется как равенство значений. Тождественность значений типа произведения или объединения определяется как покомпонентная тождественность. Тождественность значений φ и φ′ предикатного типа с наборами аргументов d и результатов e определяется как тождественность предикатов: ∀d ∀e (LS(φ(d: e)) ≡ LS(φ′(d: e))).

Определение (4.11) свойства согласованности уточняет ранее приведенное определение (2.4). Значением переменной предикатного типа является имя предиката или массива (см. разд. 4.8 и 4.9). Порождение нового значения предикатного типа реализуется через конструктор предиката (или массива) и сопровождается заведением уникального имени предиката или массива. Поэтому разные срабатывания H(x: y) могут дать в результате разные имена предикатов (или массивов), причем эти предикаты тождественны.

[15] Если H – переменная предикатного типа, то будем считать, что H входит в набор x.

4.4. Семантика вызова предиката

Пусть имеется вызов предиката

$$A(z{:}\ u)\ . \qquad (4.12)$$

Здесь A есть имя предиката или имя переменной предикатного типа; z – возможно пустой набор имен переменных; u – непустой набор имен переменных. Для случая, когда A – имя предиката, рассмотрим соответствующее определение предиката A:

$$A(x{:}\ y) \equiv K(x{:}\ y)\ , \qquad (4.13)$$

где x и y – наборы переменных, причем все переменные различны; K(x: y) – оператор. Пусть вызов (4.12) находится в правой части определения предиката B. Допустим, что определение предиката B исполняется в секции q. Тогда переменные наборов z и u принадлежат секции q. Исполнение вызова A(z: u) будем записывать на метаязыке в виде вызова процедуры runCall(q, A(z: u)), тело которой представлено ниже:

```
s = newSect(A);                    (4.14)
s[x] := q[z];
runStat(s, K(x: y));
q[u] := s[y]  .
```

В теле runCall сначала создается секция s для переменных определения A. Значения аргументов z вызова A(z: u) копируются в аргументы x определения A. В рамках секции s исполняется оператор K(x: y), что на метаязыке представлено вызовом runStat(s, K(x: y)). Наконец, результаты y копируются в результаты вызова u. Если в вызове (4.12) A есть имя переменной предикатного типа, то в теле (4.14) вместо первого оператора исполняется s = newSect(q[A]).

Поскольку секции s и q различны, а исполнение runStat(s, K(x: y)) не меняет содержимое секции q, при завершении исполнения тела (4.14) будет истинным соотношение

$$x = z \ \&\ u = y\ . \qquad ,\qquad (4.15)$$

Если предикат A является базисным, то вызов A(z: u) считается элементарным оператором абстрактного процессора, и для его исполнения не требуется исполнять тело (4.14). Тем не менее, и в этом случае исполнение вызова A(z: u) будем обозначать через runCall(q, A(z: u)). Исполнение вызова A(z: u) для случая, когда A является переменной предикатного типа, а значением A – массив, определено в разд. 4.9.

Лемма 4.2. Пусть имеется определение (4.13) для предиката A. Тогда $Cons(K) \Rightarrow Cons(A)$. Отметим, что здесь Cons(A) относится к произвольному вызову A(z: u) для некоторых наборов z и u.

Доказательство. Пусть истинно Cons(K). Докажем истинность Cons(A). Зафиксируем значения наборов z и u. Пусть истинно A(z: u). Тогда в соответствии с определением (4.13) истинна формула K(z: u). Из этого следует, что существует исполнение K(z: u) для набора z, которое завершается получением набора u. При завершении исполнения тела (4.14) в соответствии с (4.15) справедливы равенства x = z и u = y. Поэтому существует исполнение K(x: y), завершающееся получением y. Тогда исполнение последовательности (4.14), а значит и исполнение A(z: u), завершается получением значений набора u, что доказывает первую часть свойства Cons(A).

Пусть исполнение A(z: u) для набора z завершается получением набора u. Тогда завершается исполнение тела (4.14), в частности исполнение K(x: y) завершается получением y. Поскольку в соответствии с (4.15) реализуются равенства x = z и u = y, то исполнение K(z: u) завершается вычислением набора u. Из Cons(K) следует, что K(z: u) истинно. Тогда из определения (4.13) следует истинность A(z: u), что доказывает вторую часть свойства Cons(A). □

4.5. Оператор суперпозиции

Оператор суперпозиции является правой частью следующего определения предиката:

$$A(x\colon y) \equiv B(x\colon z);\ C(z\colon y)\ . \qquad (4.16)$$

Здесь x, y и z – произвольные непересекающиеся наборы переменных, причем набор x может быть пустым. Если B или C есть имя переменной предикатного типа, оно входит в набор x. Переменные набора z являются *локальными* переменными. Логическая семантика оператора суперпозиции представлена следующим образом:

$$LS(B(x\colon z);\ C(z\colon y)) \ \cong\ \exists z.\ (B(x, z)\ \&\ C(z, y))\ . \qquad (4.17)$$

Допустим, определение предиката A исполняется в рамках секции s. Секция s состоит из переменных наборов x, y и z. Исполнение правой части определения (4.16) соответствует оператору runStat(s, K(x: y)) в (4.14) и определяется следующим образом:

$$\begin{aligned} &runCall(s, B(x\colon z));\\ &runCall(s, C(z\colon y))\ . \end{aligned} \qquad (4.18)$$

Таким образом, алгоритм вычисления суперпозиции состоит в последовательном исполнении вызовов предикатов B и C. Значения переменных набора z, вычисленные при исполнении предиката B, используются при вычислении предиката C.

Лемма 4.3. Cons(B) & Cons(C) ⇒ Cons(H), где H обозначает B(x: z); C(z: y).

Доказательство. Пусть истинно Cons(B) и Cons(C). Докажем истинность Cons(H). Зафиксируем значения x и y. Пусть истинна H(x: y), т. е. истинна правая часть (4.17). Для некоторого z будут истинны B(x: z) и C(z: y). Из Cons(B) и Cons(C) следует, что исполнение B(x: z) завершается получением набора z, а исполнение C(z: y) – вычислением набора y. Поэтому исполнение операторов (4.18), а значит и H(x: y), завершается вычислением y. Это доказывает первую часть истинности Cons(H).

Допустим, исполнение H(x: y) завершается вычислением y. Докажем истинность H(x: y). Исполнение пары операторов (4.18) завершается вычислением y. В частности, завершается первый оператор runCall(s, B(x: z)). Тогда существует некоторый набор z, являющийся результатом исполнения. Таким образом, для некоторого z исполняются вызовы B(x: z) и C(z: y). Из Cons(B) и Cons(C) следует истинность B(x: z) и C(z: y). Следовательно, истинна правая часть (18) и H(x: y). □

4.6. Параллельный оператор

Параллельный оператор является правой частью следующего определения предиката:

$$A(x\colon y, z) \equiv B(x\colon y)\ ||\ C(x\colon z)\ . \qquad (4.19)$$

Здесь, x, y и z – произвольные непересекающиеся наборы переменных, причем набор x может быть пустым. Если B или C есть имя переменной предикатного типа, то оно входит в набор x. Логическая семантика параллельного оператора представлена следующим образом:

$$LS(B(x\colon y)\ ||\ C(x\colon z)) \ \cong\ B(x, y)\ \&\ C(x, z)\ .$$

Допустим, определение предиката A исполняется в рамках секции s. Секция s состоит из переменных наборов x, y и z. Исполнение правой части определения (4.19) соответствует оператору runStat(s, K(x: y)) в (4.14) и определяется следующим образом:

$$\begin{aligned} &runCall(s, B(x\colon y))\ ||\\ &runCall(s, C(x\colon z))\ . \end{aligned}$$

Алгоритм вычисления параллельной композиции || на метаязыке слагается из независимого вычисления указанных вызовов предикатов B и C. Поскольку эти два вычисления между собой не взаимодействуют, они могут проводиться параллельно. Эффект параллельной композиции равен конъюнкции эффектов композируемых операторов, т. е. B(x, y) & C(x, z).

Лемма 4.4. Cons(B) & Cons(C) ⇒ Cons(H), где H обозначает B(x: y) || C(x: z).

4.7. Условный оператор

Условный оператор является правой частью следующего определения предиката:

$$A(b, x{:}\ y) \equiv \textbf{if (}b\text{) } B(x{:}\ y) \textbf{ else } C(x{:}\ y)\ . \qquad (4.20)$$

Здесь x и y – произвольные непересекающиеся наборы переменных, причем набор x может быть пустым, b – переменная логического типа, не встречающаяся в наборах x и y. Если B или C есть имя переменной предикатного типа, то оно входит в набор x. Логическая семантика условного оператора представлена следующим образом:

$$LS(\textbf{if (}b\text{) } B(x{:}\ y) \textbf{ else } C(x{:}\ y)) \ \cong\ (b \Rightarrow B(x, y)) \ \&\ (\neg b \Rightarrow C(x, y))\ . \qquad (4.21)$$

Допустим, определение предиката A исполняется в рамках секции s. Секция s включает переменную b и переменные наборов x и y. Исполнение правой части определения (4.20) соответствует оператору runStat(s, K(x: y)) в (4.14) и определяется следующим образом:

$$\textbf{if } (s[b])\ runCall(s, B(x{:}\ y)) \textbf{ else } runCall(s, C(x{:}\ y))\ . \qquad (4.22)$$

Таким образом, алгоритм исполнения условного оператора зависит от значения переменной b. Если значение b истинно, исполняется вызов предиката B. В противном случае исполняется вызов предиката C.

Лемма 4.5. Cons(B) & Cons(C) ⇒ Cons(H), где H(b, x: y)
обозначает **if (**b) B(x: y) **else** C(x: y).

Доказательство. Пусть истинно Cons(B) и Cons(C). Докажем истинность Cons(H). Зафиксируем значения b, x и y. Пусть истинна H(b, x: y). Тогда в соответствии с (4.21) истинны формулы b ⇒ B(x: y) и ¬b ⇒ C(x: y). Рассмотрим случай, когда значение b истинно. Тогда истинна формула B(x: y). Из Cons(B) следует, что исполнение вызова B(x: y) завершается получением набора y. Поэтому исполнение оператора (4.22), а значит и H(b, x: y), завершается вычислением y. Это доказывает первую часть свойства Cons(H) для истинного значения b. Доказательство в случае ложного значения b проводится аналогично.

Допустим, исполнение H(b, x: y) завершается вычислением y, т. е. исполнение оператора (4.22) завершается вычислением y. Докажем истинность H(b, x: y). Пусть b истинно. Тогда исполнение вызова B(x: y) завершается получением набора y. Из Cons(B) следует истинность B(x: y). Тогда истинны формулы b ⇒ B(x: y) и ¬b ⇒ C(x: y), поскольку b истинно. Следовательно, истинна правая часть (4.21) и H(b, x: y). Доказательство истинности H(b, x: y) в случае ложного значения b проводится аналогично. □

4.8. Конструктор предиката

Конструктор предиката является базисным предикатом ConsPred(x, B: A), где x – возможно пустой набор переменных; B – имя предиката; A – имя переменной предикатного типа. Значением переменной A является новый предикат, создаваемый при исполнении вызова предиката ConsPred. **Ограничение**: в качестве B нельзя использовать ConsPred или ConsArray. Логическая семантика предиката ConsPred дается определением

$$LS(ConsPred(x, B{:}\ A)) \ \cong\ \forall y \forall z.\ (LS(A(y{:}\ z)) \equiv LS(B(x, y{:}\ z)))\ . \qquad (4.23)$$

Здесь x, y и z – произвольные непересекающиеся наборы переменных, причем наборы x и y могут быть пустыми. Переменная A имеет предикатный тип PRED(Y: Z), см. (4.7), где Y и Z – наборы типов, соответствующие наборам переменных y и z.

Допустим, вызов предиката ConsPred(x, B: A) находится в правой части определения некоторого предиката, исполняемого в рамках секции s. Секция s включает набор x и переменную A. Исполнение вызова ConsPred(x, B: A) является частным случаем исполнения вызова runCall(s, ConsPred(x, B: A)) (см. разд. 4.4) и реализуется следующим оператором:

$$s[A] := newDef(s[x], B)\ . \qquad (4.24)$$

Функция newDef строит новое определение предиката:

$$A_x(y{:}\ z) \equiv =(x^{\sim}{:}\ t);\ B(t, y{:}\ z)\ . \qquad (4.25)$$

Здесь $x^{\sim} = s[x]$ – значение набора x; A_x – новое имя предиката, отличное от имен других предикатов в программе. Имя A_x является результатом вызова newDef. Оператор $=(x^{\sim}: t)$ присваивает набор значений $x^{\sim}$ набору локальных переменных t.

Мы не фиксируем, каким образом абстрактный процессор исполняет оператор $=(x^{\sim}: t)$. Например, процессор может создавать дополнительные определения, строящие изображения значений $x^{\sim}$. Существует другой способ реализации вызова ConsPred(x, B: A), в котором переменной A присваивается кортеж (s[x], B).

Лемма 4.6. Cons(ConsPred).

Доказательство. Зафиксируем значения $x = x_0$ и $A = A_0$. Пусть истинно ConsPred(x, B: A). Докажем, что исполнение ConsPred(x, B: A) завершается значением $A = A_x$, тождественным A_0, т. е. $A_x \equiv A_0$. Из истинности ConsPred(x, B: A) следует истинность правой части (4.23). Тогда истинна формула $A_0(y: z) \equiv B(x_0, y: z)$, однозначно определяющая предикат A_0. Рассмотрим исполнение вызова ConsPred(x, B: A), т. е. оператора (4.24). Получаем $A = A_x$, где A_x определяется формулой (4.25). Поскольку $x^{\sim} = s[x] = x_0$, то

$$A_x(y: z) \equiv =(x_0: t);\ B(t, y: z) \equiv t = x_0\ \&\ B(t, y: z) \equiv B(x_0, y: z)\ . \qquad (4.26)$$

Таким образом, $A = A_x \equiv A_0$, что доказывает первую половину леммы.

Допустим, исполнение ConsPred(x, B: A) завершается вычислением $A = A_0$ при $x = x_0$. Докажем истинность ConsPred(x, B: A), т. е. истинность правой части (4.23). Из (4.24) следует $A = A_x$, и далее $A_0 = A_x$. Из тождества (4.26) следует $A_0(y: z) \equiv B(x_0, y: z)$. Поскольку $A = A_0$ и $x = x_0$, получаем истинность правой части (4.23). □

Лемма 4.7. Пусть имеется вызов ConsPred(x, B: A), причем Cons(B) истинно. Тогда истинно Cons(A).

Доказательство. Значением переменной A является A_x с определением (4.25). Поскольку свойство Cons истинно для оператора $=(x^{\sim}: t)$, то по лемме 4.3 оно реализуется также для правой части (4.25). Далее, по лемме 4.2 истинно $Cons(A_x)$, а значит – и Cons(A). □

4.9. Конструктор массива

Конструктор массива является базисным предикатом ConsArray(x, B: A), где x – возможно пустой набор переменных, B – имя предиката, отличное от ConsPred и ConsArray. Значением переменной A предикатного типа (4.7) является новый массив, создаваемый при исполнении вызова предиката ConsArray. Логическая семантика предиката ConsArray дается определением

$$LS(ConsArray(x, B: A)) \cong \forall y \forall z.\ (LS(A(y: z)) \equiv LS(B(x, y: z)))\ . \qquad (4.27)$$

Здесь x, y и z – произвольные непересекающиеся наборы переменных, причем набор x может быть пустым, а y и z не пусты. Значения y называются индексами, а z – элементами массива. Переменная A имеет предикатный тип PRED(Y: Z), где Y и Z – наборы типов, соответствующие наборам переменных y и z. Типы набора Y являются конечными. Произвольный массив $\varphi \in PRED(Y: Z)$ определен для любого набора индексов, т. е. истинно $\forall y \in Y\ \exists z \in Z\ \varphi(y: z)$.

Логическая семантика ConsArray (4.27) и ConsPred (4.23) идентична, однако отличается исполнение. Пусть вызов ConsArray(x, B: A) находится в правой части определения некоторого предиката, исполняемого в рамках секции s. Секция s включает набор x и переменную A, значением которой является массив $A^{\sim}$, т. е. $s[A] = A^{\sim}$. Массив $A^{\sim}$ кодирует предикат A следующим образом:

$$A(y: z) \equiv A^{\sim}[y] = z\ . \qquad (4.28)$$

Исполнение вызова ConsArray(x, B: A) является частным случаем исполнения вызова runCall(s, ConsPred(x, B: A)) (см. разд. 4.4) и реализуется следующими операторами:

```
s[A] = newArray(Y, Z);                                   (4.29)
forAll y∈Y do runCall(s, B(x, y: s[A][y])) end .
```

Первый оператор создает в памяти массив $A^{\sim}$ как значение типа PRED(Y: Z). Массив становится значением переменной A в секции s. Оператор **forAll** реализует полный перебор наборов индексов, принадлежащих набору типов Y, и для каждого набора индексов y исполняет тело оператора **forAll**, причем исполнение тела для разных y может проводиться параллельно. Для каждого набора y значение результата вызова предиката B присваивается элементу $A^{\sim}[y]$. Допустим, эффект оператора S(y: z) определяется формулой F(y). Тогда эффект оператора **forAll** $y \in Y$ **do** S(y: z) **end** есть

$$\forall y \in Y.\ F(y)\ . \qquad (4.30)$$

Лемма 4.8. Cons(A). **Доказательство** непосредственно следует из соотношения (4.28). □

Лемма 4.9. Допустим, предикат B(x, y: z) определяет однозначную всюду определенную функцию по x и y, т. е. истинно условие

$$\forall x\ \forall y \in Y\ \exists! z\ B(x, y: z)\ . \qquad (4.31)$$

Тогда Cons(B) $\Rightarrow$ Cons(ConsArray).

Доказательство. Пусть истинно Cons(B). Зафиксируем x. В соответствии с (4.30) после исполнения оператора **forAll** из (4.29) будет истинной формула: $\forall y \in Y.\ B(x, y: A^{\sim}[y])$. Пусть z – тот единственный набор, для которого истинно B(x, y: z) в соотношении (4.31). Тогда $A^{\sim}[y] = z$. Из (4.28) следует истинность A(y: z). В итоге, после исполнения тела **forAll** будет истинно $\forall y \in Y.\ B(x, y: z)\ \&\ A(y: z)$. Поскольку z единственно для B(x, y: z) и A(y: z), то будет истинным $\forall y \in Y.\ B(x, y: z) \equiv A(y: z)$. Дальнейшее доказательство очевидно. □

Допустим, в секции s исполняется вызов предиката A(u: v), где A является переменной предикатного типа, а значением A является массив. Секция s содержит переменную A и наборы u и v. Исполнение вызова A(u: v) реализуется следующим оператором:

$$s[v] := s[A]\ [u]\quad .$$

Для исполнения вызова A(u: v) в разд. 4.4 принято обозначение runCall(s, A(u: v)).

4.10. Программа

Программа на языке ССР состоит из конечного набора определений предикатов. Для каждого определения правой частью является оператор суперпозиции (4.16), параллельный оператор (4.19) или условный оператор (4.20). Первичными конструкциями, используемыми для построения операторов, являются вызовы предикатов. Набор определений предикатов программы является *замкнутым*: вызываемые предикаты являются либо базисными, либо имеет определение в программе.

Имя определяемого предиката должно быть отлично от имен базисных предикатов, имен других определяемых предикатов, а также от имен типов, используемых в программе. Если в программе используется подмножество типа (4.3), то предикат, определяющий подмножество, должен быть базисным или определяемым.

Исполнение программы заключается в исполнении некоторого вызова предиката A(z: u), где A – имя предиката, определяемого в программе; z и u – наборы переменных. Исполнение программы реализуется следующей последовательностью операторов:

```
s = newSect(z, u);                    (4.32)
input(s[z]);
runCall(s, A(z: u));
output(s[u])   .
```

Здесь s – секция, состоящая из переменных наборов z и u. Оператор input реализует ввод некоторых значений набора z в секции s. Далее в секции s исполняется вызов предиката A(z: u).

Предикаты могут быть аргументами и результатами предикатов. Создание нового предиката реализуется вызовом генератора CONS(x, B: A), где CONS обозначает базисный пре-

дикат ConsPred или ConsArray, x – возможно пустой набор переменных, являющихся фиксируемыми аргументами предиката с именем B (см. разд. 4.8 и 4.9). Генератор создает предикат и присваивает его переменной A предикатного типа.

Для упрощения языка ССР будем считать, что в качестве B в генераторе нельзя использовать переменную предикатного типа. Если все же B – переменная предикатного типа, то можно построить эквивалентную программу следующим образом. Допустим, в некотором определении предиката имеется генератор CONS(y, C**:** B). Заменим этот генератор вызовом G(y **:** B, F) и добавим в программу два определения:

G(y **:** B, F) ≡ CONS(y, C**:** B) || CONS(y, H**:** F) ;
H(y, x**:** A) ≡ CONS(y, x, C**:** A) .

Если далее B передается аргументом некоторого вызова, то необходимо дополнительным параметром передать F. Наконец, мы можем заменить генератор CONS(x, B**:** A) эквивалентным вызовом F(x**:** A).

Другое упрощение в языке ССР: имя определяемого или базисного предиката может встречаться в качестве аргумента только в вызовах ConsPred и ConsArray. Если все же имя предиката встречается в вызове другого предиката, его можно эквивалентно заменить переменной предикатного типа.

Пусть в программе имеется вызов A(z**:** u), где A – переменная предикатного типа. Значением переменной A является предикат, генерируемый некоторым вызовом CONS(x, B**:** A). В качестве представителя значения удобнее использовать предикат B, который назовем *заместителем* для данного вхождения переменной A. Введем обозначение subs(A) для множества заместителей переменной A. Определим систему правил построения множества subs(A). Допустим, вызов A(z**:** u) входит в правую часть определения D(...A**:** ...) ≡ ... A(...). В данном обозначении опускается вхождение другого вызова в правой части, опускаются вхождения аргументов и результатов предиката D, кроме аргумента A, причем считается, что A может находиться в любой позиции списка аргументов. Далее, пусть имеется вызов D(...C**:** ...), где позиция переменной C соответствует позиции A. Тогда заместители переменной C являются также заместителями A. Данное правило действует также и для определения вида D(...A**:**) ≡ ... H(...A**:**...). В итоге имеем:

Правило S1. Пусть имеется определение D(...A**:** ...) ≡ ... A(...) или D(...A**:** ...) ≡ ... H(...A**:**...) и вызов D(...C**:** ...). Тогда заместители переменной C являются заместителями A.

При наличии генератора CONS(y, D**:** E) возможны *неявные вызовы* предиката D. Если D ∈ subs(F) и имеется вызов F(...C**:** ...), то при его исполнении может быть вызвано определение предиката D. Поэтому в правиле S1 вместе с явными вызовами D(...C**:** ...) следует рассматривать и неявные вызовы F(...C**:** ...), причем позиция C сдвигается на число фиксированных аргументов генератора. Через Calls(D) обозначим множество явных и неявных вызовов предиката D. Если в соответствии с правилом S1 для вхождения A(...) будет обнаружен заместитель B для переменной A, то вызов A(...) включается в множество Calls(B). Сформулируем оставшиеся правила.

Правило S2. Пусть имеется определение D(...) ≡ C(...**:** A...); H(...A**:** ...) или D(...) ≡ C(...**:** A...); A(...). Тогда всякий заместитель переменной A для вхождения C(...**:** A...) является также заместителем A для вхождения H(...A**:** ...) или A(...). Если для вхождения A(...) обнаружен заместитель B, то вызов A(...) включается в Calls(B).

Правило S3. Пусть имеется определение H(...**:** A...) ≡ ... C(...**:** A...), а также вызов H(...**:** E...) или неявный вызов F(...**:** E...). Тогда всякий заместитель E для вхождения H(...**:** E...) (или F(...**:** E...)) является также заместителем A для вхождения C(...**:** A...).

Правило S4. Пусть имеется вызов CONS(x, B**:** A). Тогда subs(A) = {B}.

Определим алгоритм построения заместителей переменных предикатного типа для произвольной программы. В начальный момент каждому вхождению любой переменной

предикатного типа сопоставим пустое множество заместителей; для каждого определяемого предиката B множество Calls(B) определим набором явных вызовов B. На каждом шаге алгоритма по всей программе срабатывают правила S1–S4 для определений предикатов и вызовов, удовлетворяющих описанным образцам; вызовы предикатов выбираются из текущих множеств Calls. Алгоритм завершается после конечного числа шагов при стабилизации множеств заместителей по всем вхождениям предикатных переменных.

Лемма 4.10. Имеется вызов A(z**:** u), где A – переменная предикатного типа. Допустим, значение A получено исполнением конструктора CONS(x, B**:** C), а затем передано в A через параметры вызовов. Тогда B ∈ subs(A).

Замечание. Для вхождения A(z**:** u) не все элементы subs(A) достижимы при исполнении, поскольку, например, вызов A(z**:** u) может находиться внутри условного оператора.

Лемма 4.11. Имеется вызов A(z**:** u), где A – переменная предикатного типа. Тогда (∀B∈subs(A). Cons(B)) ⇒ Cons(A). **Доказательство** следует из лемм 4.7, 4.8, 4.10. □

4.11. Рекурсивные определения предикатов

Для определяемых предикатов B и C отношение depend(B, C) обозначает *непосредственную зависимость* B от C, если в правой части определения B имеется вызов предиката C. Предикат B *определяется через* предикат C, def(B, C), если существуют предикаты D_1, …, D_n ($n > 1$) такие, что $B = D_1$, $C = D_n$ и $depend(D_j, D_{j+1})$ ($j = 1, \ldots, n-1$).

Определение предиката B *рекурсивно*, если истинно отношение def(B, B). Для рекурсивно определяемого предиката B *рекурсивным кольцом* называется набор предикатов rec(B) = {C | def(B, C) & def(C, B)}. Очевидно, что если C ∈ rec(B), то rec(B) = rec(C).

Возможны более сложные формы рекурсии. Предикат B *косвенно зависит* от C, если в правой части определения B имеется вызов предиката H(x, A**:** ...) и C ∈ subs(A). Вызов H(x, A**:** ...) определяет также зависимость между предикатами H и C, так как предикат C вызывается в правой части определения H. Косвенная зависимость предикатов может оказаться источником рекурсии. В частности, при наличии генератора CONS(x, B**:** A) рекурсия для предиката B может быть обусловлена вызовом предиката A в правой части определения предиката B. В данной работе запрещаются сложные формы рекурсии, поскольку для построения реальных алгоритмов в них нет необходимости. Далее мы рассматриваем рекурсию на базе непосредственной зависимости предикатов.

График предиката B(x**:** y) есть Gr(B) = {(x, y) | LS(B(x**:** y))}. Как следствие, LS(B(x**:** y)) ≡ (x, y) ∈ Gr(B). Представим графики оператора суперпозиции (4.16), параллельного оператора (4.19) и условного оператора (4.20), используя определения логической семантики операторов:

$$GrS(P, Q) = Gr(B(x{:}\ z);\ C(z{:}\ y)) = \{(x, y) \mid \exists z.\ (x, z) \in P\ \&\ (z, y) \in Q\}\ ; \quad (4.33)$$

$$GrP(P, Q) = Gr(B(x{:}\ y)\ ||\ C(x{:}\ z)) = \{(x, y, z) \mid (x, y) \in P\ \&\ (x, z) \in Q\}\ ; \quad (4.34)$$

$$GrC(P, Q) = Gr(\mathbf{if}\ (b)\ B(x{:}\ y)\ \mathbf{else}\ C(x{:}\ y)) = \{(b, x, y) \mid b\ \&\ (x, y) \in P\} \cup \{(b, x, y) \mid \neg b\ \&\ (x, y) \in Q\}\ . \quad (4.35)$$

Здесь графики P = Gr(B) и Q = Gr(C).

Определим график для рекурсивного предиката, а через график – логическую семантику. Пусть имеется рекурсивное кольцо предикатов $A_1, A_2, \ldots, A_n$:

$$A_i(x_i{:}\ y_i) \equiv K_i(x_i{:}\ y_i);\ i = 1\ldots n;\ n > 0, \quad (4.36)$$

где x_i и y_i – наборы переменных, K_i – оператор суперпозиции (4.16), параллельный оператор (4.19) или условный оператор (4.20). В операторах K_i могут встречаться вызовы предикатов $A_1, A_2, \ldots, A_n$, а также вызовы других предикатов, не принадлежащих кольцу и определяемых вне кольца.

Обозначим графики предикатов $A_1, A_2, \ldots, A_n$ через $G_1, G_2, \ldots, G_n$ соответственно. По системе определений (4.36) строится система уравнений для графиков предикатов:

$$G = V(G)\ , \tag{4.37}$$

где $G = (G_1, G_2,..., G_n)$ – вектор графиков предикатов; $V = (Gr(K_1), Gr(K_2),..., Gr(K_n))$ – вектор графиков операторов. Здесь $Gr(K_j)(x_j: y_j)$ $(j = 1,..., n)$ рассматривается как $Gr(K_j)(G_1, G_2,..., G_n)$; $Gr(K_j)$ есть GrS, GrP или GrC.

Вектор G определяет множество значений вида $(x_1, y_1, x_2, y_2, ..., x_n, y_n)$ и является графиком, называемым *вектор-графиком*. График G_j есть j-я компонента (проекция) вектор-графика G, т. е. $G_j = pr(j, G)$.

Отношение включения "$\subseteq$", определенное на множестве вектор-графиков, является полной решеткой [4; 5]. Минимальным элементом решетки является пустой график $\varnothing$. Для произвольных вектор-графиков H и L истинно утверждение

$$H \subseteq L \equiv \forall i=1..n.\ H_i \subseteq L_i\ . \tag{4.38}$$

Отношение "$\subseteq$" для каждой из компонент вектор-графика является полной решеткой.

Рассмотрим цепь векторов-графиков $\{G^m\}_{m\geq 0}$, определяемую следующим образом:

$$G^0 = \varnothing,\ G^{m+1} = V(G^m),\ m \geq 0\ . \tag{4.39}$$

Естественно ожидать, что предел последовательности $\{G^m\}_{m\geq 0}$ (если он существует) даст нам неподвижную точку – решение системы $G = V(G)$. Построение неподвижной точки рассмотрим на примере следующей программы для вычисления факториала натурального числа n:

$$Factorial(n: f) \equiv \textbf{if}\ (n = 0)\ f = 1\ \textbf{else}\ f = n * Factorial(n - 1)\quad .$$

Запишем эту программу на языке исчисления вычислимых предикатов.

A0(n: f) ≡ A1(: c0, c1); A2(n, c0, c1: f) [16] ;
A1(: c0, c1) ≡ ConsIntZero(: c0) || ConsIntOne(: c1);
A2(n, c0, c1: f) ≡ =(n, c0: b); A3(n, c1, b: f);
A3(n, c1, b: f) ≡ **if** (b) =(c1: f) **else** A4(n, c1: f);
A4(n, c1: f) ≡ -(n, c1: n1); A5(n, n1: f);
A5(n, n1: f) ≡ A0(n1: f1); *(n, f1: f) .

Вектор-график рекурсивного кольца $G = (G0, G2, G3, G4, G5)$; предикат A1 не входит в кольцо. Проследим несколько шагов построения неподвижной точки, где шаг соответствует переходу от G^m к G^{m+1}. Вначале $G^0 = \varnothing$, т. е. каждый из шести графиков пуст. Подставляя эти значения графиков в правые части определений, получим значения графиков на первом шаге, остающиеся пустыми, кроме $G3^1$, которому присваивается первая альтернатива условного оператора. Анализируя предыдущие определения, получим значение графика $G3^1$: 0, 1, **true** $\to$ 1. На втором шаге изменится только значение графика G2, на третьем – G0. В итоге получаем следующую цепочку модификаций, в которой на каждом шаге мы указываем лишь изменившиеся компоненты вектора G :

$G3^1$: 0, 1, **true** $\to$ 1;
$G2^2$: 0, 0, 1 $\to$ 1;
$G0^3$: 0 $\to$ 1;
$G5^4$: 1, 0 $\to$ 1 – параметр $n1 = 0$ как результат подстановки $G0^3$;
$G4^5$: 1, 1 $\to$ 1;
$G3^6$: 0, 1, **true** $\to$ 1; $G3^6$: 1, 1, **false** $\to$ 1;
$G2^7$: 0, 0, 1 $\to$ 1; $G2^7$: 1, 0, 1 $\to$ 1;
$G0^8$: 0 $\to$ 1; $G0^8$: 1 $\to$ 1; и т. д.

Лемма 4.12. Пусть $\{H^m\}_{m\geq 0}$ – возрастающая цепь вектор-графиков. Тогда для наименьшей верхней грани цепи (см. разд. 3.1) справедливо равенство

$$\cup_{m\geq 0} H^m = (\cup_{m\geq 0} H_1^m, ..., \cup_{m\geq 0} H_n^m)\ .$$

Лемма 4.13. Пусть $\{H^m\}_{m\geq 0}$ – возрастающая цепь вектор-графиков и $H^{\sim} = \cup_{m\geq 0} H^m$. Пусть $w \in H^{\sim}$, где w – набор вида $(x_1, y_1, x_2, y_2, ..., x_n, y_n)$. Тогда $\exists k\ \forall m \geq k.\ w \in H^m$.

[16] Чтобы не загромождать пример, используется оператор суперпозиции более общего вида по сравнению с (4.18).

Доказательство от противного. Допустим, что для всех k набор w не принадлежит вектор-графику H^k. Определим вектор-график $H^{\#}$, совпадающий с $H^{\sim}$ всюду, за исключением набора w, который ему не принадлежит. Предикат $H^{\#}$ является верхней гранью последовательности $\{H^m\}_{m\geq 0}$, причем $H^{\#} \subseteq H^{\sim}$ и $H^{\#} \neq H^{\sim}$, а тогда грань $H^{\sim}$ не является минимальной, что приводит к противоречию. □

Замечание. Лемма верна и для возрастающей цепи обычных графиков $\{H_j^m\}_{m\geq 0}$.

Лемма 4.14. Вектор-функция V, составленная из непрерывных функций V_i (i = 1, ..., n), является непрерывной. Понятие непрерывности функций определено в разд. 3.1.

Доказательство. Пусть $\{H^m\}_{m\geq 0}$ – возрастающая цепь вектор-графиков. В соответствии с леммой 4.12 имеет место цепочка равенств:

$$\cup_{m\geq 0}V(H^m) = \cup_{m\geq 0}(V_1(H^m), ..., V_n(H^m)) = (\cup_{m\geq 0}V_1(H^m), ..., \cup_{m\geq 0}V_n(H^m)) =$$
$$= (V_1(\cup_{m\geq 0}H^m), ..., V_n(\cup_{m\geq 0}H^m)) = V(\cup_{m\geq 0}H^m). \square$$

Лемма 4.15. Функции GrS, GrP и GrC (4.33–4.35), определяющие соответственно графики оператора суперпозиции, параллельного оператора и условного оператора, непрерывны относительно вхождений графиков P и Q.

Доказательство. Докажем непрерывность функции GrS(P, Q) для оператора суперпозиции. Пусть $\{H^m\}_{m\geq 0}$ – возрастающая цепь вектор-графиков. Здесь H вырождается в набор (P, Q). Необходимо доказать, что $GrS(\cup_{m\geq 0}(P, Q)^m) = \cup_{m\geq 0} GrS((P, Q)^m)$. Это эквивалентно $GrS(\cup_{m\geq 0}P^m, \cup_{m\geq 0}Q^m) = \cup_{m\geq 0} GrS(P^m, Q^m)$ при условии, что $\{P^m\}_{m\geq 0}$ и $\{Q^m\}_{m\geq 0}$ – возрастающие цепи графиков предикатов. Итак, требуется доказать:

$$\{(x, y) \mid \exists z.\ (x, z) \in \cup_{m\geq 0}P^m \ \&\ (z, y) \in \cup_{m\geq 0}Q^m\} = \cup_{m\geq 0}\{(x, y)|\exists z.\ (x, z) \in P^m \ \&\ (z, y) \in Q^m\}.$$

Пусть (x, y) принадлежит множеству в левой части. Тогда для некоторого z_0 истинны $(x, z_0) \in \cup_{m\geq 0}P^m$ и $(z_0, y) \in \cup_{m\geq 0}Q^m$. В соответствии с леммой 4.13 существует такое k, что $(x, z_0) \in P^k$ и $(z_0, y) \in Q^k$. Далее, очевидно, (x, y) принадлежит правой части равенства.

Нетрудно показать, что $\{(x, y)|\exists z.\ (x, z) \in P^m \ \&\ (z, y) \in Q^m\}_{m\geq 0}$ – возрастающая цепь графиков. Допустим, (x, y) принадлежит наименьшей верхней грани этой цепи, т. е. правой части равенства. Принадлежность (x, y) множеству в правой части доказывается применением леммы 4.13.

Доказательство непрерывности функций GrP и GrC проводится аналогично.□

График GrC(P, Q) условного оператора **if** (b) B(x**:** y) **else** C(x**:** y) не является непрерывным относительно вхождения переменной b [17]. Поэтому вхождение b не может быть источником рекурсии. **Ограничение**: допустим, значение переменной b является результатом вызова E(...**:** b...). Предикат, правой частью которого является условный оператор, и предикат E не могут входить в одно и то же рекурсивное кольцо.

Допустим, система (4.36) определений кольца предикатов не использует сложные формы рекурсии и удовлетворяет отмеченному ограничению. Из лемм 4.14 и 4.15 следует, что вектор-график операторов V в системе уравнений G = V(G) для графиков кольца предикатов является непрерывным. В соответствии с теоремой Клини о неподвижной точке (см. разд. 3.1 и [4, 7]) решение системы G = V(G) есть $G = lfp(V) = \cup_{m\geq 0}G^m$, где $G^0 = \varnothing$, $G^{m+1} = V(G^m)$, $m \geq 0$ (т. е. цепь (40)), lfp(V) – наименьшая неподвижная точка (least fixed point) вектор-графика операторов V.

Пусть рекурсивный предикат D(z**:** u) принадлежит кольцу (4.36), т. е. $D = A_j$ для некоторого j. Логическая семантика предиката D определяется следующим образом:

$$LS(D(z{:}\ u)) \cong (z, u) \in pr(j, \cup_{m\geq 0}G^m)\ . \qquad (4.40)$$

Систему (4.36) определений кольца предикатов запишем в векторном виде: $A \equiv K(A)$, где $A = (A_1,\ A_2, \ldots,\ A_n)$; $K = (K_1,\ K_2, \ldots,\ K_n)$. Определим цепь векторов предикатов $\{A^m\}_{m\geq 0}$:

$$A^0 \equiv \Phi,\ A^{m+1} \equiv K(A^m),\ m \geq 0, \qquad (4.41)$$

[17] Не обеспечивается даже монотонность для $\neg b\ \&\ (x, y) \in Q$.

где $\Phi \cong (F, F, ..., F)$ – вектор тотально ложных предикатов. Цепь $\{A^m\}_{m\geq 0}$ соответствует цепи (4.39) вектор-графиков $\{G^m\}_{m\geq 0}$, поскольку $G^m = (Gr(A^m_1), Gr(A^m_2), \ldots, Gr(A^m_n))$.

Рассмотрим подробнее структуру рекурсивного кольца предикатов. Пусть предикат D принадлежит кольцу и имеет определение, в правой части которого вызываются предикаты B и C. Тогда один из предикатов (или оба) принадлежат кольцу. В противном случае предикат D не будет принадлежать кольцу. В соответствии с определением последовательности $\{A^m\}_{m\geq 0}$ и цепи (4.39) $D^0 = F$. Далее, $D^1 = F$, если D имеет определение в виде оператора суперпозиции или параллельного оператора, поскольку конъюнкция двух вызовов предикатов, из которых один ложный, дает ложный предикат. Если D имеет определение в виде условного оператора, а вызываемые предикаты B и C оба принадлежат кольцу, то также имеем $D^1 = F$. Если во всех условных операторах, используемых в определениях предикатов кольца, оба вызываемых предиката принадлежат кольцу, то решением системы (4.36) является набор тождественно ложных предикатов. Поэтому в дальнейшем будем считать, что в системе (4.36) имеется по крайней мере один предикат с определением в виде условного оператора, причем в правой части один вызываемый предикат принадлежит кольцу, а другой – нет.

Рекурсивное кольцо предикатов (4.36) представим в виде

$$A \equiv K(A_1, A_2, \ldots, A_n, E_1, E_2, \ldots, E_s) \quad . \qquad (4.42)$$

Здесь $E_1, E_2, \ldots, E_s$, $s > 0$, – предикаты, используемые в правых частях определений (4.36), но не принадлежащих данному кольцу.

Лемма 4.16. Предположим, что предикаты $E_1, E_2, \ldots, E_s$, используемые в системе (4.42), обладают свойством согласованности. Тогда рекурсивные предикаты $A_1, A_2, \ldots, A_n$ кольца (4.42) обладают свойством согласованности.

Доказательство. Пусть $D(u: v)$ – рекурсивный предикат кольца, т. е. $D = A_j$ для некоторого j. Докажем истинность Cons(D). Зафиксируем значения наборов u и v.

Допустим, $D(u: v)$ истинно, и докажем, что исполнение вызова $D(u: v)$ на выбранных значениях набора u завершается вычислением значений набора v. В соответствии с (4.40) $(u, v) \in pr(j, G)$. Существует $w \in G$ такой, что $(u, v) = pr(j, w)$. По лемме 4.13 существует k такой, что $w \in G^k$. Доказательство реализуется индукцией по k. Рассмотрим случай $k = 1$, т. е. $w \in G^1$. Единственная возможность: правая часть определения D имеет вид условного оператора **if** (b) $B(u: v)$ **else** $C(u: v)$, причем значение b выбирает предикат B или C, не принадлежащий кольцу; в противном случае $D(u: v)$ будет ложным. Поскольку выбранный предикат (B или C) обладает свойством согласованности, то его исполнение (а значит, и исполнение D) завершается вычислением набора u, что завершает доказательство для $k = 1$. При доказательстве для произвольного $k > 1$ будем использовать индукционное предположение: если предикат кольца $P(t: q)$ истинен для наборов t и q, и (t, q) является соответствующим поднабором некоторого $w1 \in G^m$ при $m < k$, то исполнение предиката P на наборе t завершается набором q. Рассмотрим случай, когда правая часть определения D имеет вид оператора суперпозиции $B(u: z); C(z: v)$. Истинность $D(u: v)$ определяет истинность $\exists z.\ B(u: z)\ \&\ C(z: v)$. Допустим, предикат B принадлежит кольцу и $B = A_i$. Тогда для некоторого z_0 истинно $(u, z_0) \in pr(i, G^{k-1})$, иначе в соответствии с определением (4.33) для GrS получим: $(u, v) \notin pr(j, G^k)$, что приводит к противоречию. В соответствии с индукционным предположением, исполнение $B(u: z_0)$ завершается получением набора z_0. Нетрудно доказать истинность $C(z_0, v)$. Далее, если предикат C принадлежит кольцу, то аналогичным образом, набор (z_0, v) должен входить в G^{k-1} для предиката C и, по индуктивному предположению, исполнение вызова $C(z_0, v)$ завершается вычислением набора v. Как следствие, исполнение $D(u: v)$ завершается набором v, что доказывает первую часть истинности Cons(D). Если предикат B или C не принадлежит кольцу, т. е. находится среди $E_1, E_2, \ldots, E_s$, то используется свойство согласованности этих предикатов. Доказательство для случаев, когда предикат D определяется в виде параллельного или условного оператора, является более простым.

Допустим, что для фиксированных значений u исполнение вызова D(u: v) завершается выбранными значениями набора v. Докажем истинность D(u: v). Совокупность исполняемых вызовов предикатов имеет форму дерева вызовов. Вершина дерева – вызов предиката с конкретными значениями аргументов и результатов вызова. Начальной вершиной является вызов D(u: v). Листья дерева – вызовы предикатов $E_1, E_2,..., E_s$. Два вызова связаны дугой, если второй вызов исполняется в теле определения предиката для первого вызова. Доказательство проводится индукцией по высоте k дерева. Пусть высота k = 1. Это возможно лишь в случае, когда правая часть определения D имеет вид условного оператора **if** (b) B(u: v) **else** C(u: v), а предикат (B или C), соответствующий исполняемому вызову, является одним из $E_1, E_2,..., E_s$. Из согласованности этих предикатов следует истинность исполняемого вызова, а значит, и истинность D(u: v).

Будем использовать следующее индукционное предположение: если вызов P(t: q) является вершиной дерева вызовов, высота m поддерева, определяемого вызовом P(t: q), меньше k, и исполнение вызова на наборе t завершается вычислением набора q, то P(t: q) истинно. Рассмотрим случай, когда правая часть определения D имеет вид оператора суперпозиции B(u: z); C(z: v). Допустим, что исполнение вызова B(u: z) (при исполнении D(u: v)) завершилось вычислением набора z_1. Если B является одним из $E_1, E_2,..., E_s$, то вызов B(u: z_1) является истинным. Допустим, B принадлежит кольцу. Вызов B(u: z_1) определяет поддерево в дереве вызовов с высотой m = k - 1. В соответствии с индукционным предположением, вызов B(u: z_1) является истинным. Аналогично доказывается истинность вызова C(z_1: v). Как следствие, истинна формула ∃z. B(u: z) & C(z: v), что определяет истинность D(u: v). Доказательство для случаев, когда предикат D определяется в виде параллельного или условного оператора, проводится аналогично. □

Теорема 4.1. Допустим, что при наличии в программе вызова вида ConsArray(x, B: A) предикат B определяет однозначную всюду определенную функцию. Допустим, что для всякого другого базисного предиката φ языка ССР реализуется свойство согласованности Cons(φ). Пусть имеется программа на языке ССР, и ее исполнение реализуется вызовом предиката D(u: v). Тогда истинно Cons(D).

Доказательство. Сначала доказательство проводится для программы, в которой нет переменных предикатного типа.

Рассмотрим случай, когда программа не содержит рекурсивно определяемых предикатов. Пусть предикат D имеет определение в виде оператора суперпозиции (4.16), параллельного оператора (4.19) или условного оператора (4.20). В соответствии с леммами 4.3, 4.4 и 4.5 достаточно установить свойство согласованности для предикатов B и C, вызываемых в правой части определения. Если эти предикаты базисные, то для них это свойство является условием теоремы. Если один из них определяемый, то его полное замкнутое определение представляется частью программы меньшего размера, что позволяет применить доказательство по индукции.

Допустим, программа на языке ССР содержит несколько рекурсивных колец предикатов. Пусть кольцо α определяется системой (4.42). Кольцо α *зависит* от кольца β, если один из предикатов $E_1, E_2,..., E_s$ принадлежит другому кольцу β. Рекурсивные кольца образуют граф. Вершинами являются кольца, а дугами – отношения зависимости колец. Допустим, что имеется цикл в графе колец. Нетрудно доказать, что предикаты разных колец, находящихся на цикле, должны принадлежать одному кольцу. Следовательно, граф не имеет циклов и является деревом.

Докажем свойство согласованности для предикатов произвольного рекурсивного кольца α, определяемого системой (4.42). Рассмотрим различные пути от α до листьев дерева колец. Пусть k – максимальная длина пути по этому набору путей. Доказательство проводится индукцией k. При k = 0 предикаты $E_1, E_2,..., E_s$ не являются рекурсивно определяемыми и, следовательно, обладают свойством согласованности. По лемме 4.16 предикаты

кольца обладают свойством согласованности. Предположим, что свойство согласованности доказано для колец с длиной пути $m = k - 1$, и докажем его для кольца α. Рассмотрим предикат E_j ($j = 1, \ldots, s$). Пусть E_j принадлежит рекурсивному кольцу β. Из кольца α имеется дуга в кольцо β. Поэтому максимальная длина пути для β по крайней мере на единицу меньше, чем k. По индуктивному предположению предикаты кольца β обладают свойством согласованности. Таким образом, предикаты $E_1, E_2, \ldots, E_s$ обладают свойством согласованности. Свойство согласованности предикатов кольца α следует из леммы 4.16.

Рассмотрим общий случай, когда программа Π может содержать переменные предикатного типа. Обозначим через SUBS(Π) набор предикатов, являющийся объединением множеств заместителей для всех переменных предикатного типа в программе Π. Для набора предикатов M из Π обозначим через Π[M] минимальную программу, являющуюся частью Π и содержащую определения предикатов набора M. Пусть $\Pi_0 = \Pi$, $\Pi_{j+1} = \Pi_j[\text{SUBS}(\Pi_j)]$, $j = 0, 1, 2, \ldots$. Докажем, что при некотором k программа $\Pi_k \neq \varnothing$, а $\Pi_{k+1} = \varnothing$. Допустим обратное, т. е. существует такое k, что $\Pi_k \neq \varnothing$ и $\Pi_{k+1} = \Pi_k$. Пусть Π_k построено на базе заместителей $B_1, B_2, \ldots, B_n$; $n > 0$. Тогда Π_k содержит вызовы генераторов $\text{CONS}(x, B_1 : A_1)$, …, $\text{CONS}(x, B_n : A_n)$. Допустим, $\text{CONS}(x, B_1 : A_1)$ встречается в программе $\Pi_k[\{B_1\}]$. В этой программе должен встречаться вызов $C(\ldots)$, где C – переменная предикатного типа и значение C получено от A_1 через параметры вызовов; если это не так, то генератор становится ненужным и может быть исключен из программы. Поскольку вызов $C(\ldots)$ доступен при исполнении тела предиката B_1, то получаем рекурсивный вызов B_1 через вызов $C(\ldots)$– это сложная форма рекурсии, которая запрещена. Поэтому $\text{CONS}(x, B_1 : A_1)$ не может встречаться в программе $\Pi_k[\{B_1\}]$. Пусть он встречается в $\Pi_k[\{B_2\}]$. Тогда из тела предиката B_2 через некоторый косвенный вызов $C(\ldots)$ вызывается предикат B_1. Если $\text{CONS}(x, B_2 : A_2)$ встречается в $\Pi_k[\{B_1\}]$, то получим сложную форму рекурсии. Поэтому генератор $\text{CONS}(x, B_2 : A_2)$ может быть доступен только из третьего предиката, например, B_3. Продолжая анализ для предикатов B_3 и далее, получим цепочку предикатов, которая неизбежно замкнется, что приведет к сложной форме рекурсии. Получили противоречие.

Итак, $\Pi_k \neq \varnothing$ и $\Pi_{k+1} = \varnothing$. Программа Π_k не содержит переменных предикатного типа. Для этого случая теорема доказана. Тогда предикаты $B_1, B_2, \ldots, B_n$, входящие в SUBS(Π_{k-1}), обладают свойством согласованности. В соответствии с леммой 4.11 любой вызов вида $C(\ldots)$ в программе Π_{k-1}, где C – переменная предикатного типа, обладает свойством согласованности. Доказательство теоремы, представленное выше, может быть повторено для программы Π_{k-1}, поскольку согласно принятым ограничениям вызов вида $C(\ldots)$ не может участвовать в рекурсии. Доказательство теоремы для произвольной программы Π_i, $i = k - 1, \ldots, 0$, проводится по индукции. □

4.12. Однозначность предикатов

Применимость правил серии L доказательства корректности программы в соответствующих леммах базируется на однозначности операторов, используемых в определениях предикатов. Покажем, что однозначность операторов гарантируется в случае однозначности базисных предикатов, используемых в определениях программы.

Лемма 4.17. Оператор суперпозиции (4.16), параллельный оператор (4.19) или условный оператор (4.20) является однозначным, если вызываемые в операторе предикаты B и C являются однозначными.

Лемма 4.18. Пусть имеется рекурсивное кольцо предикатов (4.36). Кроме рекурсивных предикатов $A_1, A_2, \ldots, A_n$, в правых частях определений кольца предикатов используются предикаты $E_1, E_2, \ldots, E_s$. Предположим, что предикаты $E_1, E_2, \ldots, E_s$ являются однозначными. Если аргумент определения кольца имеет предикатный тип, то предикат, являющийся значением аргумента, считается однозначным. Тогда рекурсивные предикаты $A_1, A_2, \ldots, A_n$ кольца (4.36) являются однозначными.

Доказательство. Из-за ограничений на сложные формы рекурсии предикат, являющийся значением аргумента, не может входить в кольцо предикатов. Поэтому можно считать, что такой предикат находится среди $E_1, E_2, \ldots, E_s$.

Пусть $D(u: v)$ – рекурсивный предикат кольца, т. е. $D = A_j$ для некоторого j. Допустим, истинны $D(u: v1)$ и $D(u: v2)$. Необходимо доказать, что $v1 = v2$. В соответствии с леммой 4.13 существует m, при котором $D^m(u: v1)$ и $D^m(u: v2)$ истинны. Поэтому достаточно доказать, что каждый элемент A^m цепи $\{A^m\}_{m\geq 0}$, определенной в (4.41), является вектором однозначных предикатов. Доказательство проводится индукцией по m.

Элемент A^0 является вектором однозначных предикатов, поскольку тотально ложный предикат F является однозначным. Допустим, по индуктивному предположению, A^{m-1} является вектором однозначных предикатов. Докажем это свойство для A^m. В правой части каждого определения из A^m используется оператор суперпозиции (4.16), параллельный оператор (4.19) или условный оператор (4.20). Вызываемые предикаты в правой части: $A^{m-1}{}_1, A^{m-1}{}_2, \ldots, A^{m-1}{}_n$ и $E_1, E_2, \ldots, E_s$ – однозначны. Однозначность компонент A^m следует из леммы 4.17. □

Базисные предикаты ConsPred и ConsArray не являются однозначными. В двух разных исполнениях вызова ConsPred(x, B: A) (при совпадающих x и B) в качестве значения переменной A будут получены разные имена. Однако при этом предикаты, обозначаемые разными именами, будут тождественны. Это обеспечивает однозначность вызовов предиката A при условии, что предикат B является однозначным. Аналогично будет однозначным предикат, порождаемый вызовом предиката ConsArray.

Лемма 4.19. В соответствии с правилами S1–S4 построения заместителей любой вызов C(...) в программе, где C – переменная предикатного типа, является однозначным, если для каждого вызова конструктора ConsPred(x, B: A) или ConsArray(x, B: A) предикат B является однозначным.

Теорема 4.2. Допустим, всякий базисный предикат языка ССР, кроме ConsPred и ConsArray, является однозначным на всем множестве значений аргументов. Пусть имеется программа на языке ССР, и ее исполнение реализуется вызовом предиката D(u: v), причем в наборе v нет переменных предикатного типа. Тогда предикат D является однозначным.

Доказательство. Сначала доказательство проводится для программы, в которой нет переменных предикатного типа.

Рассмотрим случай, когда программа не содержит рекурсивно определяемых предикатов. Если предикат D – базисный, то его однозначность гарантируется условием теоремы. Пусть предикат D имеет определение в виде оператора суперпозиции (4.16), параллельного оператора (4.19) или условного оператора (4.20). В соответствии с леммой 4.17 достаточно установить однозначность предикатов B и C, вызываемых в правой части определения. Если эти предикаты базисные, то их однозначность является условием теоремы. Если один из них – определяемый, то его полное замкнутое определение представляется частью программы меньшего размера, что позволяет применить доказательство по индукции.

Допустим, в программе имеются рекурсивно определяемые предикаты. В соответствии с анализом, проведенным при доказательстве теоремы 4.1, совокупность рекурсивно определяемых предикатов образует дерево колец. Для колец, являющихся листьями дерева, однозначность следует из леммы 4.18. Доказательство однозначности предикатов остальных колец проводится индукцией по длине пути в дереве колец. Отметим, что доказательство

легко обобщается для случая, когда в программе имеются переменные предикатного типа, значениями которых являются однозначные предикаты.

Допустим, в программе Π имеются переменные предикатного типа. Обозначим через SUBS(Π) набор предикатов, являющийся объединением множеств заместителей для всех переменных предикатного типа в программе Π. Для набора предикатов M из Π обозначим через $\Pi[M]$ минимальную программу, являющуюся частью Π и содержащую определения предикатов набора M. Пусть $\Pi_0 = \Pi$, $\Pi_{j+1} = \Pi_j[SUBS(\Pi_j)]$, $j = 0, 1, 2, \ldots$. В теореме 4.1 доказано, что для некоторого k программа $\Pi_k \neq \varnothing$, а $\Pi_{k+1} = \varnothing$. Программа Π_k не содержит переменных предикатного типа. Однозначность предикатов программы Π_k доказана выше. Тогда предикаты $B_1, B_2, \ldots, B_n$, входящие в SUBS(Π_{k-1}), являются однозначными. В соответствии с леммой 4.19 любой вызов вида $C(...)$ в программе Π_{k-1}, где C – переменная предикатного типа, является однозначным. Доказательство теоремы, представленное выше, обобщается для программы Π_{k-1}, поскольку согласно принятым ограничениям вызов вида $C(...)$ не может участвовать в рекурсии. Доказательство теоремы для произвольной программы Π_i, $i = k - 1, \ldots, 0$, проводится по индукции. □

Список литературы

1. *Backus J.* Can programming be liberated from the von Neumann style? A. Functional Style and Its Algebra of Programs // Communications of the ACM. 1978. Vol. 21 (8). P. 613–641.
2. *Hoare C. A. R.* О структурной организации данных // Структурное программирование. М.: Мир, 1975. С. 98–197.
3. *Tennent R. D.* Semantics of Programming Languages. Pentice Hall, 1991. 236 p.
4. *Крицкий С. П.* Трансляция языков программирования: синтаксис, семантика, перевод: Учеб. пособие. Р.-на-Дон.: РГУ, 2005.
URL: http://public.uic.rsu.ru/~skritski/scourses/Transl/Langs1.htm.
5. *Гретцер Г.* Общая теория решеток: Пер. с англ. 1982. 456 с.
7. *Лавров С. С.* Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001. 320 с.
8. *Milner R.* A Calculus of Communicating Systems // Lecture Notes in Computer Science. 1980.Vol. 92.
9. *Hoare C. A. R.*. Communicating Sequential Processes. Prentice-Hall. 1985.

5. Семантика языка предикатного программирования. Методы доказательства корректности предикатных программ

Язык CCP (язык исчисления вычислимых предикатов; см. разд. 4) определяет минимальное полное ядро для построения произвольного (чистого) языка функционального программирования в виде иерархической системы обозначений на базе конструкций ядра. Язык предикатного программирования P (Predicate programming language) конструируется как результат последовательного расширения цепочки языков (слоев):

$$CCP = P0 \subset P1 \subset P2 \subset P3 \subset \ldots \subset P.$$

Каждый очередной слой языка определяется в виде набора производных конструкций от предыдущего слоя.

Каждая новая конструкция C очередного слоя вводится как обозначение некоторой языковой композиции K предыдущего слоя. Логическая семантика LS(C) новой конструкции получается тождественным логическим преобразованием формулы LS(K), а операционная семантика – эквивалентным преобразованием метапрограммы для композиции K. Такой механизм построения логической и операционной семантики конструкции C гарантирует их согласованность в соответствии с определением (4.11). В большинстве случаев правила доказательства корректности конструкции C удается построить преобразованием соответствующих правил для композиции K. Предлагается две серии правил: R и L. Серия R определяет правила доказательства истинности постусловия из программы по формуле (2.10). Серия L определяет правила вывода программы из спецификации по теореме 2.1 тождества спецификации и программы; в этой теореме требуется однозначность программы и спецификации. Однозначность операторов программы гарантируется в случае однозначности базисных предикатов в соответствии с теоремой 4.2.

5.1. Язык P1: подстановка определения предиката на место вызова

Определим *подстановку определения предиката* A(x: y) ≡ K(x: y) (см. (4.2)) на место вызова A(t: z), где x, y, t и z – наборы переменных. Результатом подстановки является конструкция { K(t: z) }, называемая *блоком*, которая вставляется в программу на место вызова A(t: z). Конструкция K(t: z) получается из K(x: y) заменой всех вхождений переменных наборов x и y на соответствующие переменные наборов t и z. Здесь и далее мы полагаем, что для любого определения предиката параметры и локальные переменные имеют уникальные имена, не встречающиеся в других определениях программы. Кроме того, если K(x: y) – оператор суперпозиции, то при построении блока { K(t: z) } дополнительно проводится переименование локальных переменных, обеспечивающее их уникальность. С учетом данных соглашений замена x и y на t и z в конструкции K(x: y) не приведет к коллизиям. Определим

$$LS(\{\ K(t: z)\ \}) \cong\ LS(K(t: z))\ . \qquad (5.1)$$

Поскольку A(t: z) ≡ K(t: z), то LS({ K(t: z) }) ≡ LS(A(t: z)).

Исполнение блока { K(t: z) } в секции s определяется на метаязыке вызовом runBlock(s, { K(t: z) }). Тело процедуры runBlock представлено оператором

$$runStat(s, K(t: z))\ . \qquad (5.2)$$

Исходя из операционной семантики вызова предиката (4.14) нетрудно показать, что исполнение вызова A(t: z) эквивалентно исполнению блока { K(t: z) }. Таким образом, программа П′, получаемая из программы П подстановкой определения предиката на место вызова, эквивалентна программе П: исполнение любого предиката программы П′ на фиксированном наборе аргументов дает тот же результат, что и в программе П.

Язык программы, получающийся многократным произвольным применением подстановок определений предикатов на место вызовов, обозначим через P1. Язык P1 является

расширением языка ССР. В языке P1 правой частью определения предиката является оператор суперпозиции (4.16), параллельный оператор (4.19) или условный оператор (4.20), причем в данных операторах на месте вызовов предикатов B и C может также находиться конструкция «блок». Блок может содержать блоки внутри себя. По построению конструкции языка P1 (операторы и блоки), полученные подстановками определений предикатов на место вызовов, обладают свойством согласованности.

5.2. Язык P2: оператор суперпозиции и параллельный оператор общего вида

Операторы { A(...); B(...) }; C(...) и A(...); { B(...); C(...) } являются эквивалентными, т. е. суперпозиция обладает свойством ассоциативности. Это доказывается на основе операционной семантики оператора суперпозиции (4.18). Аналогичным образом устанавливается ассоциативность параллельной композиции, т. е. эквивалентность { A(...) || B(...) } || C(...) и A(...) || { B(...) || C(...) }. Свойство ассоциативности позволяет опускать внутренние фигурные скобки. Введем в языке P2 оператор суперпозиции и параллельный оператор общего вида: $A_1(...); A_2(...); ... ; A_n(...)$ и $A_1(...) || A_2(...) || ... || A_n(...)$ для $n > 1$.

Рассмотрим оператор суперпозиции общего вида со спецификацией в виде предусловия P(x) и постусловия Q(x, y):

$$P(x)\ \{B1(x{:}\ z_1);\ B2(z_1{:}\ z_2)\ ;...;\ Bj(z_{j-1}{:}\ z_j)\ ;...;\ Bn(z_{n-1}{:}\ y)\}\ Q(x, y)\ ^{18}\ . \qquad (5.3)$$

Здесь $x, z_1, z_2, \ldots, z_{n-1}, y$ – различные непересекающиеся наборы переменных; B1, B2, …, Bn обозначают вызовы предикатов или блоки языка P1 со спецификациями (предусловиями и постусловиями) $P_{B1}(x)$, $Q_{B1}(x, z_1)$, $P_{B2}(z_1)$, $Q_{B2}(z_1, z_2)$, …, $P_{Bn}(z_{n-1})$, $Q_{Bn}(z_{n-1}, y)$. Логическая семантика оператора суперпозиции общего вида определяется на основе логической семантики суперпозиции двух операторов (4.17):

$$LS(B1(x{:}\ z_1);\ B2(z_1{:}\ z_2)\ ;...;\ Bj(z_{j-1}{:}\ z_j)\ ;...;\ Bn(z_{n-1}{:}\ y)) \equiv$$
$$\exists z_1,z_2,...,z_{n-1}.\ LS(B1(x{:}\ z_1))\ \&\ LS(B2(z_1{:}\ z_2))\ \&...\&\ LS(Bj(z_{j-1}{:}\ z_j))\ \&...\&\ LS(Bn(z_{n-1}{:}\ y))\ . \quad (5.4)$$

Допустим, определение предиката A исполняется в рамках секции s. Секция s состоит из переменных наборов x, y и $z_1, z_2, ..., z_{n-1}$. Исполнение оператора (5.3) реализуется на метаязыке оператором $runStat(s, B1(x{:}\ z_1);\ ...;\ Bn(z_{n-1}{:}\ y))$ в (4.14) и определяется следующим образом:

$$\begin{aligned} &runCallBlock(s, B1(x{:}\ z_1)); \qquad (5.5)\\ &runCallBlock(s, B2(z_1{:}\ z_2)); .....;\\ &runCallBlock(s, Bj(z_{j-1}{:}\ z_j)); .....;\\ &runCallBlock(s, Bn(z_{n-1}{:}\ y))\quad . \end{aligned}$$

Здесь runCallBlock обозначает runCall, если вторым параметром является вызов предиката, или runBlock в случае блока.

Определим правила серии L, гарантирующие корректность оператора суперпозиции общего вида (5.3).

Правило LS3. $P(x) \vdash P_{B1}(x)$.

Правило LS4. $P(x)\ \&\ Q(x, y)\ \&\ Q_{B1}(x, z_1) \vdash$

$$\exists!z_2,z_3,...,z_{n-1}.\ P_{B2}(z_1)\ \&\ Q_{B2}(z_1, z_2)\ \&...\&\ P_{Bj}(z_{j-1})\ \&\ Q_{Bj}(z_{j-1}, z_j)\ \&...\&\ P_{Bn}(z_{n-1})\ \&\ Q_{Bn}(z_{n-1}, y).$$

Лемма 5.1. Допустим, спецификация P(x) и Q(x, y) оператора суперпозиции общего вида реализуема, операторы B1, …, Bn однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LS3 и LS4 истинны, то оператор суперпозиции (5.3) является корректным.

Доказательство. Поскольку операторы $B_1, \ldots, B_n$ однозначны, то и оператор суперпозиции (5.3) является однозначным. В соответствии с (5.4) и теоремой 2.1 для доказательства леммы достаточно доказать истинность формулы

[18] Здесь и далее используется иная форма записи тройки Хоара по сравнению с {P(x)} S {Q(x, y)} в гл. 2.

$P(x)\ \&\ Q(x, y) \Rightarrow$

$\exists z_1,z_2,...,z_{n-1}.\ LS(B1(x{:}\ z_1))\ \&\ LS(B2(z_1{:}\ z_2))\ \&...\&\ LS(Bj(z_{j-1}{:}\ z_j))\ \&...\&\ LS(Bn(z_{n-1}{:}\ y))$. (5.6)

Пусть истинны $P(x)$ и $Q(x, y)$. Докажем истинность правой части (5.6). Из истинности предусловия $P(x)$ по правилу LS3 следует истинность $P_{B1}(x)$. Из корректности оператора B1 по правилу E2 следует истинность формулы $\exists z_1.\ LS(B1(x{:}\ z_1))$. Допустим, для некоторого z'_1 истинно $LS(B1(x{:}\ z'_1))$. Для оператора B1 истинны условия леммы 2.8. Поэтому истинно $Q_{B1}(x, z'_1)$. В соответствии с правилом LS4 истинна правая часть LS4 для $z_1 = z'_1$. Тогда из истинности $P_{Bj}(z_{j-1})\ \&\ Q_{Bj}(z_{j-1}, z_j)$ для $j = 3,..., n$ ($z_n = y$) по лемме 2.9 следует истинность $LS(Bj(z_{j-1}{:}\ z_j))$. Это доказывает истинность формулы (5.6). □

Определим правила серии R, гарантирующие корректность оператора суперпозиции общего вида (5.3).

Правило RS3. $P(x) \vdash P_{B1}(x)\ \&\ \forall z_1,z_2,..., z_{n-1}.$

$(Q_{B1}(x, z_1) \Rightarrow P_{B2}(z_1))\ \&\ (Q_{B2}(z_1, z_2) \Rightarrow P_{B3}(z_2))\ \&\ ...\ \&\ (Q_{Bn-1}(z_{n-2}, z_{n-1}) \Rightarrow P_{Bn}(z_{n-1}))$.

Правило RS4. $P(x)\ \&\ \exists z_1,z_2,...,z_{n-1}\ (Q_{B1}(x, z_1)\ \&\ Q_{B2}(z_1, z_2)\ \&\ ...\ \&\ Q_{Bn}(z_{n-1}, y)) \vdash Q(x, y)$.

Лемма 5.2. Пусть предусловие $P(x)$ истинно. Допустим, операторы B1, …, Bn корректны. Если правила RS3 и RS4 истинны, то оператор суперпозиции (5.3) является корректным.

Доказательство. В соответствии с формулой (2.10) достаточно доказать реализуемость $LS(B1; B2; ..., Bn)(x, y)$ и выводимость постусловия $Q(x, y)$ из $LS(B1; B2; ..., Bn)(x, y)$. Вместо $LS(B1; B2; ..., Bn)(x, y)$ рассматривается правая часть (5.4).

Из истинности предусловия $P(x)$ по правилу RS3 следует истинность правой части правила, в том числе формулы $P_{B1}(x)$. Из истинности $P_{B1}(x)$ и правила E2 следует истинность формулы $\exists z.\ LS(B1(x{:}\ z))$. Допустим, для некоторого z'_1 формула $LS(B1(x{:}\ z'_1))$ истинна. Из истинности $P_{B1}(x)$ и правила E3 следует истинность $LS(B1(x{:}\ z'_1)) \Rightarrow Q_{B1}(x, z'_1)$. Как следствие, истинно $Q_{B1}(x, z'_1)$. Из истинности правой части RS3 следует истинность формулы $\forall z_1\ (Q_{B1}(x, z_1) \Rightarrow P_{B2}(z_1))$. Как следствие, истинно $P_{B2}(z'_1)$. По правилу E2 истинна формула $\exists z_2\ LS(B2)(z'_1, z_2)$. Пусть формула истинна для z'_2. Из истинности $Q_{B2}(z_1, z_2) \Rightarrow P_{B3}(z_2)$ следует истинность $P_{B3}(z'_2)$. Аналогичным образом, по индукции доказывается истинность $LS(Bj(z_{j-1}{:}\ z_j))$ для $j = 1,..., n - 1$. Это доказывает реализуемость $LS(B1; B2; ..., Bn)(x, y)$.

Допустим, истинна правая часть (5.4). Докажем истинность постусловия $Q(x, y)$. Пусть правая часть (5.4) истинна для $z'_1, z'_2,..., z'_{n-1}$. По правилу E3 истинна формула $LS(B1(x{:}\ z'_1)) \Rightarrow Q_{B1}(x, z'_1)$ и далее – $Q_{B1}(x, z'_1)$. Истинность $Q_{B1}(x, z'_1)$ и $\forall z\ (Q_{B1}(x, z) \Rightarrow P_{B2}(z))$ влечет истинность $P_{B2}(z'_1)$. По правилу E3 истинно $LS(B2(z'_1{:}\ z'_2)) \Rightarrow Q_{B2}(z'_1, z'_2)$. Поскольку $LS(B2(z'_1{:}\ z'_2))$ истинно, то истинно $Q_{B2}(z'_1, z'_2)$. Аналогичным образом по индукции доказывается истинность $Q_{Bj}(z'_{j-1}, z'_j)$ для $j = 2, ..., n - 1$. Таким образом, истинна правая часть правила RS4, а значит, и левая, т. е. истинно постусловие $Q(x, y)$. □

Рассмотрим параллельный оператор общего вида со спецификацией в виде предусловия $P(x)$ и постусловия $Q(x, y)$:

$$P(x)\ \{B1(x{:}\ y_1)\ ||\ B2(x{:}\ y_2)\ ||...||\ Bj(x{:}\ y_j)\ ||...||\ Bn(x{:}\ y_n)\}\ Q(x, y)\ . \qquad (5.7)$$

Здесь $x, y_1, y_2, …, y_n$ – различные непересекающиеся наборы переменных; набор y объединяет в себе наборы $y_1, y_2, …, y_n$; B1, B2, …, Bn обозначают предикаты и блоки языка P1 со спецификациями (предусловиями и постусловиями) $P_{B1}(x)$, $Q_{B1}(x, y_1)$, $P_{B2}(x)$, $Q_{B2}(x, y_2)$, …, $P_{Bn}(x)$, $Q_{Bn}(x, y_n)$. Логическая семантика параллельного оператора общего вида определяется на основе логической семантики для двух операторов (см. разд. 4.6):

$LS(B1(x{:}\ y_1)\ ||\ B2(x{:}\ y_2)\ ||...||\ Bj(x{:}\ y_j)\ ||...||\ Bn(x{:}\ y_n)) \equiv$

$$LS(B1(x{:}\ y_1))\ \&\ LS(B2(x{:}\ y_2))\ \&\ ...\ \&\ LS(Bj(x{:}\ y_j))\ \&\ ...\ \&\ LS(Bn(x{:}\ y_n))\ . \qquad (5.8)$$

Допустим, определение предиката A исполняется в рамках секции s. Секция s состоит из переменных наборов $x, y_1, y_2, …, y_n$. Исполнение оператора (5.7) реализуется на метаязыке оператором $runStat(s, K(x{:}\ y))$ в (4.14) и определяется следующим образом:

runCallBlock(s, B1(x: y_1)) ||
runCallBlock(s, B2(x: y_2)) || ... ||
runCallBlock(s, Bj(x: y_j)) || ... ||
runCallBlock(s, Bn(x: y_n)) . (5.9)

Определим правила серии L, гарантирующие корректность параллельного оператора общего вида (5.7).

Правило LP4. $P(x) \vdash P_{B1}(x) \,\&\, P_{B2}(x) \,\&\, ... \,\&\, P_{Bn}(x)$.

Правило LP5. $P(x) \,\&\, Q(x, y) \vdash Q_{B1}(x, y_1) \,\&\, Q_{B2}(x, y_2) \,\&\, ... \,\&\, Q_{Bn}(x, y_n)$.

Лемма 5.3. Допустим, спецификация P(x) и Q(x, y) параллельного оператора общего вида реализуема, операторы B1, …, Bn однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LP4 и LP5 истинны, то параллельный оператор (5.7) является корректным.

Определим правила серии R, гарантирующие корректность параллельного оператора общего вида (5.7).

Правило RP3. $P(x) \vdash P_{B1}(x) \,\&\, P_{B2}(x) \,\&\, ... \,\&\, P_{Bn}(x)$.

Правило RP4. $Q_{B1}(x, y_1) \,\&\, Q_{B2}(x, y_2) \,\&\, ... \,\&\, Q_{Bn}(x, y_n) \vdash Q(x, y)$.

Лемма 5.4. Пусть предусловие P(x) истинно. Допустим, операторы B1, …, Bn корректны. Если правила RP3 и RP4 истинны, то параллельный оператор (5.7) является корректным.

Условимся, что суперпозиция «связывает» сильнее, чем параллельная композиция; т. е. вместо {A(...); B(...)} || C(...) в языке P2 будем писать A(...); B(...) || C(...). Если в операторе A(x: y); {B(z: t) || C(u: v)} набор y пересекается с набором z и не пересекается с набором u, то оператор эквивалентен {A(x: y); B(z: t)} || C(u: v) или, с учетом соглашения о приоритетах, оператору A(x: y); B(z: t) || C(u: v). Аналогичным образом оператор {A(x: y)||B(z: t)};C(u: v) эквивалентен A(x: y) || B(z: t); C(u: v), если наборы y и u не пересекаются.

5.3. Язык P2: другое обобщение оператора суперпозиции

Оператор B(x: z); C(z: y) определен как наиболее простая форма оператора суперпозиции; см. (4.16). Оператор B(x: z); C(x, z: y) является обобщением оператора суперпозиции. Применяя данное обобщение к оператору суперпозиции общего вида (5.3), получим оператор

$$P(x)\ \{B1(x: z_1);\ B2(x, z_1: z_2)\ ;...;\ Bj(x, z_{j-1}: z_j)\ ;...;\ Bn(x, z_{n-1}: y)\}\ Q(x, y)\ . \quad (5.10)$$

Логическая и операционная семантика, правила RS3, RS4, LS3, LS4 обобщаются, а леммы 5.1 и 5.2 аналогичным образом доказываются для оператора (5.10). Отметим, что оператор вида B(x: z); C(u, z: y), где набор u содержит часть переменных набора x, можно рассматривать как частный случай оператора B(x: z); C(x, z: y).

Наиболее общей формой суперпозиции двух операторов является оператор

$$A(x: t, y) \equiv P(x)\ \{B(x: z, t);\ C(x, z: y)\}\ Q(x, t, y)\ . \quad (5.11)$$

Здесь x, y, z, t – различные непересекающиеся наборы переменных, причем наборы x и t могут быть пустыми; B и C обозначают предикаты или блоки языка P1 со спецификациями (предусловиями и постусловиями) $P_B(x)$, $Q_B(x, z, t)$, $P_C(x, z)$, $Q_C(x, z, y)$.

Оператор суперпозиции (5.11) можно определить через оператор суперпозиции (4.16) и параллельный оператор (4.19) языка CCP следующим образом. Введем определения предикатов B1 и C1:

$$B1(x: x1, z, t1) \equiv P_B(x)\ \{B(x: z, t1) \,||\, x1 = x\}\ Q_B(x, z, t1) \,\&\, x1 = x\ ;$$

$$C1(x1, z, t1: y, t) \equiv P_C(x1, z)\ \{C(x1, z: y) \,||\, t = t1\}\ Q_C(x1, z, y) \,\&\, t = t1\ .$$

Поскольку B1(x: x1, z, t1); C1(x1, z, t1: t, y) ≡ B(x: z, t); C(x, z: y), то справедливо другое определение предиката A:

$$A(x: t, y) \equiv P(x)\ \{B1(x: x1, z, t1);\ C1(x1, z, t1: t, y)\}\ Q(x, t, y)\ . \quad (5.12)$$

Определение (5.12) по форме соответствует простейшему оператору суперпозиции (4.16). Подставим правую часть (5.12) в определение логической и операционной семантики (4.17, 4.18) и проведем эквивалентные преобразования. Получим логическую и операционную семантику оператора (5.11):

$$LS(B(x{:}\ z, t);\ C(x, z{:}\ y)) \equiv \exists z.\ (LS(B(x{:}\ z, t)) \ \&\ LS(C(x, z, y))) . \quad (5.13)$$

runCallBlock(s, B(x**:** z, t)); (5.14)

runCallBlock (s, C(x, z**:** y)) .

Для определения (5.12) применимы леммы 2.5 и 2.11. Применим правила RS1 и RS2 для оператора в правой части (5.12). Получим следующие правила.

Правило RS1′. $P(x) \vdash P_B(x)\ \&\ \forall x1, z, t1\ ((Q_B(x, z, t1)\ \&\ x1 = x) \Rightarrow P_C(x1, z))$.

Правило RS2′. $P(x)\ \&\ \exists x1,z,t1\ (Q_B(x1, z, t1)\ \&\ x1 = x\ \&\ Q_C(x1, z, y)\ \&\ t = t1) \vdash Q(x, t, y)$.

Эквивалентные преобразования RS1′ и RS2′ дают правила RS5 и RS6, по форме подобные правилам RS1 и RS2 для простейшего оператора суперпозиции (4.16).

Правило RS5. $P(x) \vdash P_B(x)\ \&\ \forall z, t\ (Q_B(x, z, t) \Rightarrow P_C(x, z))$.

Правило RS6. $P(x)\ \&\ \exists z\ (Q_B(x, z, t)\ \&\ (Q_C(x, z, y)) \vdash Q(x, t, y)$.

Как следствие, справедлива следующая лемма.

Лемма 5.5. Пусть предусловие P(x) истинно. Допустим, операторы B и C корректны. Если правила RS5 и RS6 истинны, то оператор суперпозиции (5.11) является корректным.

Оператор суперпозиции вида (5.11) можно обобщить для произвольного числа композируемых операторов. Обобщается логическая и операционная семантика, а также лемма 5.5.

Применим правила LS1 и LS2 для оператора в правой части (5.12). Получим следующие правила.

Правило LS1′. $P(x) \vdash P_B(x)$.

Правило LS2′. $P(x)\ \&\ Q(x, t, y)\ \&\ Q_B(x, z, t1)\ \&\ x1 = x \vdash P_C(x1, z)\ \&\ Q_C(x1, z, y)\ \&\ t = t1$.

Эквивалентные преобразования LS1′ и LS2′ дают правила LS6 и LS7.

Правило LS6. $P(x) \vdash P_B(x)$.

Правило LS7. $P(x)\ \&\ Q(x, t, y)\ \&\ Q_B(x, z, t1) \vdash P_C(x, z)\ \&\ Q_C(x, z, y)\ \&\ t = t1$.

Как следствие, справедлива следующая лемма.

Лемма 5.6. Допустим, спецификация оператора суперпозиции (5.11) реализуема, операторы B и C однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LS6 и LS7 истинны, то оператор суперпозиции (5.11) является корректным.

5.4. Язык **P3**: выражения

Наряду с предикатной (или операторной) формой представления конструкций будем использовать *функциональную* форму. Пусть имеется вызов предиката A(t**:** z), где t и z – наборы переменных. Как эквивалентную для вызова A(t**:** z) будем использовать запись в функциональном стиле: z = A(t). Конструкцию A(t) назовем *вызовом функции*. В случае, когда z – набор более чем из одной переменной, будем также использовать форму записи |z| = A(t).

Для базисных предикатов языка ССР, соответствующих стандартных арифметических операциям, вместо функциональной формы z = A(t) будем использовать традиционную инфиксную и постфиксную нотацию. Например, базисные предикаты: +(x, y**:** z), –(x, y**:** z), –(x**:** y), <(x, y**:** b) – записываются в языке P3 привычным образом: z = x + y, z = x – y, y = –x , b = x < y.

В языке P3 вводятся изображения констант. Так, вместо базисных предикатов ConsIntZero(**:** x) и ConsIntOne(**:** x) используется запись вида x = 0 и x = 1. Значение x константы целого типа по ее изображению s определяется предикатом valInt(s **:** x). Напри-

мер, x = 2089 интерпретируется как вызов valInt("2089" **:** x). Аналогичным образом определяются изображения констант для других примитивных типов.

Применение функционального стиля позволяет преобразовать оператор суперпозиции следующим образом:

$$B(x\mathbf{:}\ z);\ C(x,\ z\mathbf{:}\ y) \equiv z = B(x);\ C(x,\ z\mathbf{:}\ y) \equiv C(x,\ B(x)\mathbf{:}\ y)\ .$$

В языке P3 аргументом вызова является либо переменная, либо вызов функции. При этом вызов функции в качестве аргумента определяется указанным тождеством. Семантика вызова, имеющего вызовы в двух и более аргументах, определяется следующим тождеством:

$$\{\ A(x\mathbf{:}\ y)\ ||\ B(z\mathbf{:}\ t)\ \};\ C(y,\ t\mathbf{:}\ u) \equiv \{\ y = A(x)\ ||\ t = B(z)\ \};\ C(y,\ t\mathbf{:}\ u) \equiv C(A(x),\ B(z)\mathbf{:}\ u)\ .$$

Таким образом, вычисление аргументов вызовов реализуется параллельно.

Определим правила доказательства корректности для вызова предиката, содержащего вызовы среди аргументов. Рассмотрим определение предиката

$$D(x,\ z\mathbf{:}\ u) \equiv P(x,\ z)\ \{C(A(x),\ B(z)\mathbf{:}\ u)\}\ Q(x,\ z,\ u)\ . \qquad (5.15)$$

Пусть предикат A имеет спецификацию $P_A(x)$ и $Q_A(x, y)$, а предикат B – $P_B(z)$ и $Q_B(z, t)$. Если предикаты A и B являются корректными, то корректно следующее определение:

$$E(x,\ z\mathbf{:}\ y,\ t) \equiv P_A(x)\ \&\ P_B(z)\ \{A(x\mathbf{:}\ y)\ ||\ B(z\mathbf{:}\ t)\}\ Q_A(x,\ y)\ \&\ Q_B(z,\ t)\ .$$

Предикат D можно представить определением

$$D(x,\ z\mathbf{:}\ u) \equiv P(x,\ z)\ \{\ E(x,\ z\mathbf{:}\ y,\ t);\ C(y,\ t\mathbf{:}\ u)\}\ Q(x,\ z,\ u)\ . \qquad (5.16)$$

Пусть предикат C со спецификацией $P_C(y, t)$ и $Q_C(y, t, u)$ является корректным. Применим правила RS1 и RS2 для оператора в правой части (5.16). Получим следующие правила.

Правило RS1′. $P(x, z) \vdash P_A(x)\ \&\ P_B(z)\ \&\ \forall y,t\ (Q_A(x, y)\ \&\ Q_B(z, t) \Rightarrow P_C(y, t))$.

Правило RS2′. $P(x, z)\ \&\ \exists y,t\ (Q_A(x, y)\ \&\ Q_B(z, t)\ \&\ Q_C(y, t, u)) \vdash Q(x, z, u)$.

Допустим, операторы A и B являются однозначными; их спецификации $P_A(x)$, $Q_A(x, y)$, $P_B(z)$, $Q_B(z, t)$ также однозначны. Эквивалентные преобразования RS1′ и RS2′ с учетом однозначности предикатов и спецификаций позволяют устранить переменные y и t, что дает правила RS7 и RS8.

Правило RS7. $P(x, z) \vdash P_A(x)\ \&\ P_B(z)\ \&\ P_C(A(x), B(z))$.

Правило RS8. $P(x, z)\ \&\ Q_C(A(x), B(z), u) \vdash Q(x, z, u)$.

В итоге, справедлива следующая лемма.

Лемма 5.7. Пусть предусловие P(x, z) истинно. Допустим, операторы A, B и C корректны, операторы A и B, а также их спецификации $P_A(x)$, $Q_A(x, y)$, $P_B(z)$, $Q_B(z, t)$ однозначны. Если правила RS7 и RS8 истинны, то оператор (5.15) со спецификацией P(x, z) и Q(x, z, u) является корректным.

Для получения правил серии L применим правила LS1 и LS2 для оператора в правой части (5.16). Получим следующие правила.

Правило LS1′. $P(x, z) \vdash P_A(x)\ \&\ P_B(z)$.

Правило LS2′. $P(x, z)\ \&\ Q(x, z, u)\ \&\ Q_A(x, y)\ \&\ Q_B(z, t) \vdash P_C(y, t)\ \&\ Q_C(y, t, u)$.

Предполагая однозначность предикатов A и B, а также их спецификаций, проведем эквивалентные преобразования LS1′ и LS2′, устраняя переменные y и t. Получим следующие правила.

Правило LS8. $P(x, z) \vdash P_A(x)\ \&\ P_B(z)$.

Правило LS9. $P(x, z)\ \&\ Q(x, z, u) \vdash P_C(A(x), B(z))\ \&\ Q_C(A(x), B(z), u)$.

Справедлива следующая лемма.

Лемма 5.8. Допустим, спецификация P(x, z) и Q(x, z, u) оператора (5.15) реализуема, операторы A, B и C однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LS8 и LS9 истинны, то оператор (5.15) является корректным.

Нетрудно проверить, что леммы 5.7 и 5.8 верны для любого числа вызовов, используемых в качестве аргументов вызова предиката C, в том числе и для одного вызова.

Пусть имеется оператор суперпозиции z = a ∗ b; y = z + c. Как определено выше, операции a ∗ b и z + c являются аналогами вызовов предикатов ∗(a, b**:** z) и +(∗z, c**:** y). Применим схему подстановки вызова на место аргумента во втором вызове суперпозиции. Подстановка a ∗ b на место z в операторе y = z + c дает оператор y = (a ∗ b) + c. Здесь используется правило: подставляемая операция на место аргумента другой операции обрамляется круглыми скобками. Таким образом, приходим к известному понятию *выражения*. Использование правил приоритетов операций, позволяющих опускать круглые скобки, определяет привычную форму записи выражений; например, y = a ∗ b + c.

Переменные, изображения констант, вызовы функций и их представление в виде операций являются частными случаями понятия выражения. Таким образом, в языке Р3 позиция аргумента вызова в общем случае содержит выражение.

Имеет место тождество

C(x**:** b); **if** (b) A(x**:** y) **else** B(x**:** y) ≡ **if** (C(x)) A(x**:** y) **else** B(x**:** y) .

Учитывая это, в языке Р3 в позиции (b) условного оператора будем использовать произвольное выражение логического типа.

Язык ССР определен как бестиповый язык. При этом тип каждой переменной программы на языке ССР можно распознать, используя информацию о вхождениях переменной в программе. Язык Р также можно реализовать как бестиповый подобно языку Лисп. Однако в случае использования полиморфных операций, например «+», можно привести примеры программ, когда невозможно однозначно определить типы переменных, что предопределяет неоднозначность программы. Тем не менее, и при использовании полиморфных операций можно было бы построить бестиповый язык, введя соответствующие ограничения. Язык Р определен как типовый. Описания аргументов и результатов предикатов и локальных переменных снабжаются указанием типов в стиле языка С. Тип, математическое описание которого дается в разд. 4.2, кодируется в языке Р конструкцией ИЗОБРАЖЕНИЕ-ТИПА; см. разд. 6.7. Аргументация в пользу типового языка заключается в том, что указание типов переменных существенно повышает понимание программы.

5.5. Методы доказательства корректности рекурсивных программ

Рассмотрим следующую конкретизацию системы (4.36) определений рекурсивного кольца предикатов $A_1, A_2, \ldots, A_n$:

$$A_j(t, x_j: y_j) \equiv P_j(t, x_j)\ \{K_j(t, x_j: y_j)\}\ Q_j(t, x_j, y_j);\ j = 1\ldots n;\ n > 0\ . \qquad (5.17)$$

Здесь P_j и Q_j – предусловие и постусловие для предиката A_j; t – набор индукционных переменных; x_j, y_j – наборы переменных. На значениях набора t определен строгий частичный порядок $\sqsubset$, удовлетворяющий свойству обрыва бесконечных убывающих цепей; см. разд. 3.2. Пусть имеется предикат β(t), истинный, по меньшей мере, на любом минимальном элементе t. Предикат β(t) определяет базу индукции. Корректность определений рекурсивного кольца представлена формулой W(t) как корректность всех определений кольца:

$$W(t) \equiv \mathop{\&}_{j=1}^{n} \{ P_j(t, x_j) \Rightarrow (\exists y_j.\ LS(K_j(t, x_j: y_j))) \ \&\ (LS(K_i(t, x_j: y_j)) \Rightarrow Q_j(t, x_j, y_j)) \}. \qquad (5.18)$$

Определим правила серии R доказательства корректности определений кольца (5.17) по методу структурной и полной индукции; см. разд. 3.2.

Правило RR1. $\beta(t) \vdash W(t)$.

Правило RR2. $\neg\beta(t)\ \&\ Induct(t) \vdash W(t)$.

Здесь $Induct(t) \cong \forall u.\ u \sqsubset t \Rightarrow W(u)$. Предикат Induct(t) соответствует индукционному предположению при доказательстве по индукции и определяет корректность определений кольца (5.17) для всех значений, строго меньших t.

Лемма 5.9. Если правила RR1 и RR2 истинны, то определения кольца (5.17) корректны.

Доказательство. Конъюнкция правил RR1 и RR2 дает формулу (3.4) доказательства методом структурной индукции. □

Определим правила серии L. Выводимость программы рекурсивного кольца (5.17) из спецификаций представим формулой V(t):

$$V(t) \equiv \mathop{\&}_{j=1}^{n} \{ P_j(t, x_j) \,\&\, Q_j(t, x_j, y_j) \Rightarrow LS(K_i(t, x_j{:}\ y_j)) \} . \qquad (5.19)$$

Правила доказательства корректности определений кольца (5.17) базируется на методе структурной индукции.

Правило LR1. $\beta(t) \vdash V(t)$.

Правило LR2. $\neg\beta(t) \,\&\, Induct(t) \vdash V(t)$.

Лемма 5.10. Допустим, для каждого предиката A_j рекурсивного кольца его спецификация (в виде предусловия P_j и постусловия Q_j) является реализуемой, а правая часть K_i – однозначной. Если правила LR1 и LR2 истинны, то определения кольца (5.17) корректны.

Доказательство. Конъюнкция правил LR1 и LR2 дает формулу (3.4) для доказательства истинности V(t) методом структурной индукции. Истинность V(t) есть истинность формулы (2.16) для всех предикатов рекурсивного кольца. Таким образом, все условия теоремы 2.1 удовлетворены, и поэтому все определения предикатов рекурсивного кольца являются корректными. □

Применимость леммы 5.10 базируется на однозначности операторов, используемых в определениях рекурсивного кольца (5.17). Однозначность операторов гарантируется в случае однозначности базисных предикатов по теореме 4.2.

Проиллюстрируем технику преобразования программы на языке CCP в эквивалентную программу на языке P3 на примере программы факториала, приведенной в разд. 4.11. Сначала используем функциональный стиль записи для операций и изображения констант. Получим:

```
A0(n: f)          ≡ A1( : c0, c1); A2(n, c0, c1: f)
A1( : c0, c1)     ≡ c0 = 0 || c1 = 1
A2(n, c0, c1: f)  ≡ b = (n = c0); A3(n, c1, b: f)
A3(n, c1, b: f)   ≡ if (b) f =c1 else A4(n, c1: f)
A4(n, c1: f)      ≡ n1 = n – c1; A5(n, n1: f)
A5(n, n1: f)      ≡ A0(n1: f1); f = n ∗ f1
```

Далее подставим A1 в A0 и A5 в A4, а также преобразуем операторы суперпозиции к функциональному виду:

```
A0(n: f)          ≡ A2(n, 0, 1: f)
A2(n, c0, c1: f)  ≡ A3(n, c1, n = c0: f)
A3(n, c1, b: f)   ≡ if (b) f =c1 else A4(n, c1: f)
A4(n, c1: f)      ≡ A5(n, n – c1: f)
A5(n, n1: f)      ≡ f = n ∗ A0(n1)
```

Подставим A3 в A2, а A2 в A0, A5 в A4:

```
A0(n: f)          ≡ if (n = 0) f =1 else A4(n, 1: f)
A4(n, c1: f)      ≡ f = n ∗ A0(n – c1)
```

Наконец, подставляя A4 в A0, получим программу факториала на языке P3:

```
A0(n: f)          ≡ if (n = 0) f =1 else f = n ∗ A0(n – 1) .
```

Использование функционального стиля и особенно подстановка определения предиката на место единственного вызова существенно улучшают структуру программы, за исключением последнего случая, в котором вызов A0(n – 1) находится в теле определения A0. Опыт показывает, что подавляющее число рекурсивных колец определений предикатов на языке P состоит из единственного рекурсивного определения.

Рассмотрим частный случай кольца (5.17) в виде следующего единственного рекурсивного определения:

$A(t, x\mathbf{:}\ y) \equiv P_A(t, x)\ \{\ \mathbf{if}\ (\varphi(t))\ B(t, x\mathbf{:}\ y)\ \mathbf{else}\ C(t, x\mathbf{:}\ y)\ \}\ Q_A(t, x, y)\ .$ (5.20)

Здесь $B(t, x\mathbf{:}\ y)$ и $C(t, x\mathbf{:}\ y)$ обозначают блоки или вызовы, причем $B(t, x\mathbf{:}\ y)$ не зависит от A, а $C(t, x\mathbf{:}\ y)$ – зависит. При наличии спецификаций B и C можно было бы доказывать корректность представленного условного оператора по правилам RC1–RC4 или LC1, LC2, однако ввиду рекурсии следует использовать правила RR1, RR2. Формула корректности определения предиката (5.20) принимает вид:

$$W(t) \equiv P_A(t, x) \Rightarrow (\exists y.\ LS(K(t, x\mathbf{:}\ y))) \ \&\ (LS(K(t, x\mathbf{:}\ y)) \Rightarrow Q_A(t, x, y))\ .$$

Здесь $K(t, x\mathbf{:}\ y)$ обозначает оператор $\mathbf{if}\ (\varphi(t))\ B(t, x\mathbf{:}\ y)\ \mathbf{else}\ C(t, x\mathbf{:}\ y)$. Правила RR1, RR2 для кольца (5.20) принимают следующий вид.

Правило RR3. $\varphi(t)\ \&\ P_A(t, x)\ \vdash (\exists y.\ LS(B(t, x\mathbf{:}\ y))) \ \&\ (LS(B(t, x\mathbf{:}\ y)) \Rightarrow Q_A(t, x, y))$.

Правило RR4. $\neg\varphi(t)\ \&\ Induct(t)\ \&\ P_A(t, x)\ \vdash$

$$(\exists y.\ LS(C(t, x\mathbf{:}\ y))) \ \&\ (LS(C(t, x\mathbf{:}\ y)) \Rightarrow Q_A(t, x, y)).$$

В данных правилах истинность или ложность условия $\varphi(t)$ позволяет специализировать $K(t, x\mathbf{:}\ y)$ в виде $B(t, x\mathbf{:}\ y)$ или $C(t, x\mathbf{:}\ y)$. В итоге, справедлива следующая лемма.

Лемма 5.11. Если правила RR3 и RR4 истинны, то рекурсивное определение (5.20) корректно.

Аналогичным образом можно специализировать правила LR1, LR2 серии L. Формула корректности (5.20) принимает вид

$$V(t) \equiv P_A(t, x)\ \&\ Q_A(t, x, y) \Rightarrow LS(K(t, x\mathbf{:}\ y))\ .$$

Правила корректности получаются из правил LR1, LR2 для (5.20) применением специализации для $K(t, x\mathbf{:}\ y)$.

Правило LR3. $\varphi(t)\ \&\ P_A(t, x)\ \&\ Q_A(t, x, y)\ \vdash LS(B(t, x\mathbf{:}\ y))$.

Правило LR4. $\neg\varphi(t)\ \&\ Induct(t)\ \&\ P_A(t, x)\ \&\ Q_A(t, x, y)\ \vdash LS(C(t, x\mathbf{:}\ y))$.

Лемма 5.12. Допустим, спецификация (предусловие $P_A(t, x)$ и постусловие $Q_A(t, x, y)$) является реализуемой, а вызовы (или блоки) $B(t, x\mathbf{:}\ y)$ и $C(t, x\mathbf{:}\ y)$ однозначны. Если правила LR3 и LR4 истинны, то определение (5.20) является корректным.

Доказательство. Из однозначности $B(t, x\mathbf{:}\ y)$ и $C(t, x\mathbf{:}\ y)$ получаем однозначность $K(t, x\mathbf{:}\ y)$. Выполняется условие леммы 5.10, что доказывает корректность (5.20). □

Техника доказательства корректности определений рекурсивного кольца базируется на замене каждого рекурсивного вызова $A(u, x\mathbf{:}\ y)$ в правой части рекурсивного определения на соответствующую спецификацию $Q_A(u, x, y)$ с использованием индукционного предположения $Induct(t)$. Если выполняется условие леммы 5.10 (или 5.12), то для $u \sqsubset t$ и при истинном предусловии $P_A(u, x)$ по лемме 2.8 получаем тождество $LS(A(u, x\mathbf{:}\ y)) \equiv Q_A(u, x, y)$, что дает возможность заменить вызов $A(u, x\mathbf{:}\ y)$ предикатом $Q_A(u, x, y)$. В общем случае, когда однозначность спецификаций не гарантируется, применяется формула
$LS(A(u, x\mathbf{:}\ y)) \Rightarrow Q_A(u, x, y)$.

Пример 5.1. Программа умножения через сложение.

Докажем корректность следующего рекурсивного определения:

```
Умн(nat a, b:  nat c)  ≡
    pre a ≥ 0 & b ≥ 0
    { if (a = 0) c = 0 else c = b + Умн(a – 1, b) }
    post c = a ∗ b;
```

Доказательство. Реализуемость спецификации гарантируется, а однозначность правой части следует из теоремы 4.2. Чтобы применить лемму 5.12, достаточно доказать истинность правил LR3 и LR4. Правило LR3 принимает следующий вид:

$$LR3:\quad a = 0\ \&\ \&\ b \geq 0\ \&\ c = a * b\ \vdash c = 0.$$

Правило истинно ввиду свойства $0 * x = 0$.

Правило LR4 переписывается в виде

$$LR4:\quad a \neq 0\ \&\ Induct(a)\ \&\ a \geq 0\ \&\ b \geq 0\ \&\ c = a * b\ \vdash c = b + Умн(a - 1, b).$$

Поскольку $a \neq 0$ и $a \geq 0$, то $a - 1 \geq 0$, и истинно предусловие для вызова Умн$(a - 1, b)$. В соответствии с индукционным предположением, применяя лемму 2.8, можно заменить вызов на $(a - 1) * b$. Далее, $b + (a - 1) * b = a * b$, что доказывает истинность правила LR4 и корректность определения предиката Умн. □

Пример 5.2. Программа D(a, b**:** c) вычисления наибольшего общего делителя (НОД) положительных a и b.

Определим свойство «x является *делителем* a» следующим предикатом:

$$x \text{ – делитель } a \cong \text{divisor}(x, a) \cong \exists z \geq 0.\ x * z = a\ .$$

Предикат:

$$\text{divisor2}(a, b, c) \cong \text{divisor}(c, a)\ \&\ \text{divisor}(c, b)$$

определяет c в качестве *общего делителя* значений a и b. Свойство *наибольшего общего делителя* определяется предикатом:

$$\text{НОД}(a, b, c) \cong \text{divisor2}(a, b, c)\ \&\ \forall x.\ (\text{divisor2}(a, b, x) \Rightarrow x \leq c)\ .$$

Приведенная далее программа вычисления наибольшего общего делителя базируется на следующих известных базисных свойствах НОД:

$$a = b\ \Rightarrow \text{НОД}(a, b, a) \tag{5.21}$$

$$a < b\ \&\ \text{НОД}(a, b, c) \Rightarrow\ \text{НОД}(a, b - a, c) \tag{5.22}$$

$$\text{НОД}(a, b, c) \equiv \text{НОД}(b, a, c) \tag{5.23}$$

Из базисных свойств НОД легко доказать истинность следующих свойств:

$$\text{НОД}(a, b, c)\ \&\ a = b\ \Rightarrow c = a \tag{5.24}$$

$$\text{НОД}(a, b, c)\ \&\ a < b\ \Rightarrow\ \text{НОД}(a, b - a, c) \tag{5.25}$$

$$\text{НОД}(a, b, c)\ \&\ a > b\ \Rightarrow\ \text{НОД}(a - b, b, c) \tag{5.26}$$

Свойства (5.24-5.26) определяют *логику решения* задачи НОД. Предикатная программа вычисления НОД непосредственно переписывается из логики решения (5.24-5.26).

```
D(nat a, b:  nat c) ≡ pre a ≥ 1 & b ≥ 1
{       if (a = b) c = a
        else   if (a < b) D(a, b – a: c)
               else D(a – b, b: c)
} post НОД(a, b, c);
```

Доказательство корректности программы D. Докажем реализуемость спецификации, т. е. истинность $\exists c.\ \text{НОД}(a, b, c)$ для положительных a и b. Для a и b имеется, по крайней мере, один общий делитель – это 1. Следовательно, существует и наибольший общий делитель. Однозначность оператора в правой части определения D следует из теоремы 4.2.

Индуктивными переменными рекурсивного определения D является пара переменных a и b. На множестве положительных целых определим отношение

$$(a, b) \sqsubseteq (c, d) \cong a \leq c\ \&\ b \leq d\ .$$

Отношение $\sqsubseteq$ является частичным порядком. Строгий частичный порядок $\sqsubset$ определяется на основе $\sqsubseteq$. Единственным минимальным элементом является (1, 1). Отношение $a = b$ в условном операторе определения D является истинным на минимальном элементе и поэтому годится в качестве базиса при доказательстве по индукции. В соответствии с леммой 5.12 достаточно доказать истинность правил LR3 и LR4. Правило LR3 принимает следующий вид:

$$\text{LR3:}\quad a = b\ \&\ \&\ a \geq 1\ \&\ b \geq 1\ \&\ \text{НОД}(a, b, c)\ \vdash c = a\ .$$

Истинность правила следует из свойства (5.21) и единственности наибольшего общего делителя. Правило LR4 записывается в виде

$$\text{LR4:}\quad a \neq b\ \&\ \text{Induct}(a, b)\ \&\ a \geq 1\ \&\ b \geq 1\ \&\ \text{НОД}(a, b, c)\ \vdash$$
$$[\ a < b \Rightarrow \text{D}(a, b - a\ \mathbf{:}\ c)\ ]\ \&\ [\ \neg\ a < b \Rightarrow \text{D}(a - b, b\ \mathbf{:}\ c)\ ]\ .$$

Пусть истинно $a < b$. Докажем истинность D$(a, b - a\ \mathbf{:}\ c)$. Так как $b - a > 0$, то для данного вызова истинно предусловие. Поскольку $(a, b - a) \sqsubset (a, b)$, то в соответствии с индуктивным предположением Induct(a, b) и по лемме 2.8 вызов D$(a, b - a\ \mathbf{:}\ c)$ тождест-

венен НОД(a, b – a, c). Истинность НОД(a, b – a, c) следует из НОД(a, b, c) и свойств (5.22), (5.23).

Пусть истинно ¬ a < b. Докажем истинность D(a – b, b : c). Из условий ¬ a < b и a ≠ b следует b < a. Дальнейшее доказательство аналогично доказательству истинности первого конъюнкта левой части LR4. □

В приведенных примерах проводится следующая схема доказательства корректности рекурсивных определений вида (5.20). Сначала проверяется реализуемость спецификации и ее однозначность [19]. Проверяется, что все элементарные операции и вызываемые предикаты в правой части являются однозначными. Устанавливаются индуктивные переменные, определяется отношение частичного порядка и проверяется, что условие φ в условном операторе покрывает все минимальные элементы. Далее, полагая истинным предусловие, постусловие и условие φ, доказывается истинность первой альтернативы условного оператора. Затем для истинного предусловия, постусловия и ложного условия φ доказывается истинность второй альтернативы условного оператора. При этом для каждого рекурсивного вызова проверяется истинность предусловия для аргументов вызова. Доказывается, что аргументы, соответствующие индуктивным переменным, строго меньше в смысле выбранного отношения. Это дает возможность заменить рекурсивный вызов соответствующим постусловием.

[19] На самом деле однозначность проверять не требуется, однако для неоднозначной спецификации будет невозможно доказать истинность правил RR3 и RR4.

6. Язык предикатного программирования

6.1. Введение

Язык предикатного программирования P (**P**redicate programming language) является универсальным и может использоваться для разработки широкого класса программ. По меньшей мере, этот класс включает программы, разрабатываемые обычно на языке ФОРТРАН, для задач вычислительной математики. В целях спецификации и реализации программ для реактивных систем язык P расширен средствами для описания процессов, передачи сообщений и порождения процессов, работающих параллельно с процессом-родителем. В меньшей степени язык P пригоден для задач системного программирования, где обычно используются объектно-ориентированные средства, которых нет в ядре языка P.

По сложившейся классификации язык P следует отнести к классу языков функционального программирования. Язык P сочетает функциональный и операторный (предикатный) стили записи алгоритмов и обладает существенно большей выразительностью по сравнению с чисто функциональными языками. В этом смысле язык P ближе к языкам логического программирования. Однако в отличие от логических языков процесс вычисления программы на языке P реализуется не логическим выводом, а явным исполнением, характерным для императивных и функциональных языков.

Функциональный язык удобнее императивного для разработки алгоритма. Однако получение эффективной программы для функциональных языков проблематично даже с применением изощренной оптимизации в процессе трансляции. Поэтому после отладки программы на функциональном языке для достижения требуемой эффективности ее приходится переписывать на императивный язык. Приемущество технологии предикатного программирования в том, что она является сквозной: к предикатной программе применяется набор трансформаций с получением эффективной программы на императивном расширении языка P (см. разд. 6.11), после чего программа может конвертироваться на любой из императивных языков: C, C++, ФОРТРАН и др. Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

С помощью данных трансформаций можно получить программу предельной эффективности, однако трудно это сделать автоматически.

Язык предикатного программирования P впервые представлен в работе [1]. Полное описание языка P дается в препринте [2]. Последующие модификации и расширения языка инициированы использованием языка для описания различных алгоритмов [3–5]. Введены средства спецификации определений предикатов в виде предусловий и постусловий. Имеется опыт использования языка P в качестве языка публикации алгоритмов. Язык P расширен для спецификации и реализации реактивных систем, определяемых в виде совокупности взаимодействующих процессов [5]. Введены средства изображения процессов, операторы приема и посылки сообщений, средства динамического порождения процессов.

Все предыдущие версии языка P по стилю были ближе к языкам Паскаль и Модула-2. Последняя версия языка P, представленная в настоящем описании, по синтаксису, набору операторов и операций и другим особенностям существенно приближена к стилю языков типа C. Как следствие, язык P станет более комфортным для программирующих на языках C, C++, Java, C#. Изменения синтаксиса языка P стали причиной проведения серии серьезных модификаций во всем языке. Определенное влияние на новую версию языка P оказал

язык спецификаиций PVS [7]. Введены алгебраические типы. Последовательности заменены списками.

Синтаксис описывается на расширенном языке Бэкусовских нормальных форм (БНФ) со следующими особенностями:

- терминальные символы выделены жирным шрифтом;
- [фрагмент] – означает возможное отсутствие в синтаксическом правиле заключенного в квадратные скобки фрагмента; фрагмент определяет последовательность терминальных и нетерминальных символов;
- (фрагмент)* – определяет повторение фрагмента нуль или более раз; круглые скобки могут быть опущены, если фрагмент состоит из одного символа;
- (фрагмент)+ – определяет повторение фрагмента один или более раз;
- запись вида [:CLASS:] обозначает символ указанного класса; используются следующие классы символов:

alpha – буквенный символ, принадлежащий латинскому или русскому алфавиту; разрешаются заглавные и строчные буквы;

digit – цифра (0, 1, 2, 3, 4, 5, 6, 7, 8 или 9);

alnum – символ, принадлежащий alpha или digit;

blank – пробел или символ табуляции;

xdigit – шестнадцатеричная цифра (digit – заглавная или строчная буква от A до F);

print – символ, для которого определено начертание (т. е. символ из alnum, blank либо какой-либо другой символ, который можно отобразить).

6.2. Лексемы

Текст программы представлен в виде последовательности строк символов. Переход на новую строку эквивалентен символу «пробел». Программа может содержать комментарии, текст которых считается не принадлежащим тексту программы. Комментарий начинается парой символов /* и завершается парой символов */. Второй вид комментариев начинается с пары символов // и продолжается до конца текущей строки текста. Текст программы составляется из лексем следующего вида:

ЛЕКСЕМА ::= ИДЕНТИФИКАТОР | КЛЮЧЕВОЕ-СЛОВО | КОНСТАНТА | ОПЕРАЦИЯ | РАЗДЕЛИТЕЛЬ| МЕТКА

ИДЕНТИФИКАТОР ::= НАЧАЛО-ИДЕНТИФИКАТОРА СИМВОЛ-ИДЕНТИФИКАТОРА*

НАЧАЛО-ИДЕНТИФИКАТОРА ::= _ | [:alpha:]

СИМВОЛ-ИДЕНТИФИКАТОРА ::= _ | [:alnum:]

Идентификатор используется для именования предикатов, переменных, типов и других объектов. Использование специального имени “_” в качестве результирующей переменной вызова предиката (см. разд. 6.3.3) обозначает пустой результат, неиспользуемый в дальнейшем вычислении. Имя “_” зарезервировано в языке P – его нельзя использовать для именования объектов программы. Другие идентификаторы, начинающиеся с “_”, считаются обычными и могут использоваться в описаниях программы.

МЕТКА ::= СИМВОЛ-ИДЕНТИФИКАТОРА*

КЛЮЧЕВОЕ-СЛОВО ::=

after | array | as | bool | break | case | char | class | else | enum
| exists | extends | default | false | for | forall | formula | if | in
| inf | int | import | message | module | nan | nat | new | nil | or
| post | pre | process | predicate | queue | real | receive | set | string
| struct | subtype | switch | type | true | union | var | while | with | xor

ОПЕРАЦИЯ ::= **+** | **-** | ***** | **/** | **%** | **^** | **!** | **<<** | **>>** | **~** | **&** | **|** | **?** |
= | **<** | **>** | **<=** | **>=** | **!=** | **=>** | **<=>**

РАЗДЕЛИТЕЛЬ ::= **(** | **)** | **[** | **]** | **{** | **}** | **,** | **;** | **:** | **.** | **..** | [:blank:]

КОНСТАНТА ::= ЦЕЛАЯ-КОНСТАНТА | ВЕЩЕСТВЕННАЯ-КОНСТАНТА |
СИМВОЛЬНАЯ-КОНСТАНТА | СТРОКОВАЯ-КОНСТАНТА |
ЛОГИЧЕСКАЯ-КОНСТАНТА | КОНСТАНТА-NIL
ЦЕЛАЯ-КОНСТАНТА ::= ЦЕЛОЕ-ЧИСЛО
ЦЕЛОЕ-ЧИСЛО ::= [ЗНАК] ЦИФРЫ | **0x** ШЕСТНАДЦАТЕРИЧНЫЕ-ЦИФРЫ
ЗНАК ::= **+** | **-**
ЦИФРЫ ::= ЦИФРА+
ЦИФРА ::= [:digit:]
ШЕСТНАДЦАТЕРИЧНЫЕ-ЦИФРЫ ::= ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА+
ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА ::= [:xdigit:]
ВЕЩЕСТВЕННАЯ-КОНСТАНТА ::= [ЗНАК] ЦИФРЫ [**.** ЦИФРЫ] [ПОРЯДОК] |
[ЗНАК] **inf** | **nan**
ПОРЯДОК ::= (**e** | **E**) [ЗНАК] ЦИФРЫ

Для вещественных типов (см. разд. 6.7) определены специальные значения: **inf** (бесконечность) и **nan** (не число). Значение **nan** возникает в процессе вычисления как результат взятия квадратного корня из отрицательного числа, деления ноля на ноль, умножения ноля на бесконечность и других операций над вещественными числами, когда результат не может быть определён.

СИМВОЛЬНАЯ-КОНСТАНТА ::= **'** СИМВОЛ **'**
СИМВОЛ ::= ОБЫЧНЫЙ-СИМВОЛ | СПЕЦИАЛЬНЫЙ-СИМВОЛ |
ШЕСТНАДЦАТЕРИЧНЫЙ-КОД-СИМВОЛА
ОБЫЧНЫЙ-СИМВОЛ ::= любой символ из [:print:], кроме **'** и **"**
СПЕЦИАЛЬНЫЙ-СИМВОЛ ::= **\"** | **\'** | **** | **\0** | **\n** | **\r** | **\t**
ШЕСТНАДЦАТЕРИЧНЫЙ-КОД-СИМВОЛА ::=
\x ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА [ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА
[ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА [ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА]]]

СТРОКОВАЯ-КОНСТАНТА ::= **"** СИМВОЛ* **"**
ЛОГИЧЕСКАЯ-КОНСТАНТА ::= **true** | **false**
КОНСТАНТА-ПУСТО ::= **nil**

6.3. Предикаты

Программа состоит из набора определений предикатов.

6.3.1. Определение предиката

ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА ::= ИМЯ-ПРЕДИКАТА ОПИСАНИЕ-ПРЕДИКАТА
ИМЯ-ПРЕДИКАТА ::= ИДЕНТИФИКАТОР

Значением ОПИСАНИЯ-ПРЕДИКАТА является предикат, обозначаемый именем предиката.

ОПИСАНИЕ-ПРЕДИКАТА ::= ОПИСАНИЕ-ПРЕДИКАТА-ФУНКЦИИ |
ОПИСАНИЕ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ОПИСАНИЕ-ПРЕДИКАТА-ФУНКЦИИ ::=
ЗАГОЛОВОК-ПРЕДИКАТА [**pre** ПРЕДУСЛОВИЕ]
ТЕЛО-ПРЕДИКАТА [**post** ПОСТУСЛОВИЕ]

ЗАГОЛОВОК-ПРЕДИКАТА ::=
([ОПИСАНИЯ-АРГУМЕНТОВ]: ОПИСАНИЯ-РЕЗУЛЬТАТОВ)
ТЕЛО-ПРЕДИКАТА ::= БЛОК
ПРЕДУСЛОВИЕ ::= ФОРМУЛА
ПОСТУСЛОВИЕ ::= ФОРМУЛА

Значения аргументов предиката должны удовлетворять предусловию. По завершении исполнения тела предиката должно быть истинным постусловие, связывающее значения аргументов и результатов.

ОПИСАНИЯ-РЕЗУЛЬТАТОВ ::=
ИЗОБРАЖЕНИЕ-ТИПА [:blank:] ИМЯ-РЕЗУЛЬТАТА (**,** ИМЯ-РЕЗУЛЬТАТА)*
[**,** ОПИСАНИЯ-РЕЗУЛЬТАТОВ]
ИМЯ-РЕЗУЛЬТАТА ::= ИДЕНТИФИКАТОР | ИДЕНТИФИКАТОР **'**

Имя вида имя′ используется для именования результирующего параметра при условии, что имя описано в качестве одного из аргументов с тем же типом. В императивной программе, получаемой трансформацией предикатной программы, результат с именем имя′ склеивается с соответствующим аргументом имя. Если тип аргумента имя параметризован (см. разд. 6.7), то значение параметра для типа результата имя′ может отличаться.

ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ ::=
ИЗОБРАЖЕНИЕ-ТИПА [:blank:] ИМЯ-АРГУМЕНТА (**,** ИМЯ-АРГУМЕНТА)* |
ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА
ОПИСАНИЯ-АРГУМЕНТОВ ::=
ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ [**,** ОПИСАНИЯ-АРГУМЕНТОВ]
ИМЯ-АРГУМЕНТА ::= ИДЕНТИФИКАТОР

```
assign (int from : int to) { to = from }
main (seq (string) argv : int ret_code) { ret_code = 0 }
power (real x, int p :  real x') { x' = x^p; }
```

Пример 1. Определения предикатов

ОПИСАНИЕ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ]
ТЕЛО-ПРЕДИКАТА [ПОСТУСЛОВИЯ-ВЕТВЕЙ]
ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
([ОПИСАНИЯ-АРГУМЕНТОВ] **:**
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ **)**
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ ::=
[ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ВЕТВИ] [**#** МЕТКА-ВЕТВИ]
(**:** ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ)+
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ВЕТВИ ::= ОПИСАНИЯ-РЕЗУЛЬТАТОВ
МЕТКА-ВЕТВИ ::= МЕТКА

После двоеточия описываются параметры-результаты очередной ветви гиперфункции. Ветви гиперфункции именуются метками. Если в описании результатов ветви нет метки, то ветвь именуется целым – порядковым номером ветви начиная с 1. Исполнение гиперфункции завершается выбором одной из ветвей и вычислением значений результатов этой ветви. При этом результаты других ветвей не определены. Оператор перехода #имяВетви в теле предиката завершает исполнение гиперфункции ветвью с именем имяВетви.

ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ ::=
[ОБЩЕЕ-ПРЕДУСЛОВИЕ] (ПРЕДУСЛОВИЕ-ВЕТВИ)+
ОБЩЕЕ-ПРЕДУСЛОВИЕ ::= **pre** ПРЕДУСЛОВИЕ
ПРЕДУСЛОВИЕ-ВЕТВИ ::= **pre** МЕТКА-ВЕТВИ **:** ПРЕДУСЛОВИЕ

Отсутствие общего предусловия определяет допустимость произвольных значений аргументов в соответствии с их типами. Предусловие ветви определяет дополнительное условие на значения аргументов, при котором исполнение гиперфункции завершается выбором этой ветви. При этом значения аргументов должны удовлетворять общему предусловию. Предусловие последней ветви обычно не указывается, поскольку оно получается дополнением предусловий предыдущих ветвей.

ПОСТУСЛОВИЯ-ВЕТВЕЙ ::= (ПОСТУСЛОВИЕ-ВЕТВИ)+
ПОСТУСЛОВИЕ-ВЕТВИ ::= **post** МЕТКА-ВЕТВИ **:** ПОСТУСЛОВИЕ

Постусловие ветви связывает значения аргументов и результатов ветви. Ветвь без результатов постусловия не имеет.

6.3.2. Спецификация предиката

Предикат, используемый в программе, должен иметь определение предиката. Если предикат определяется в другом модуле программы, то предикат может быть представлен своей спецификацией.

СПЕЦИФИКАЦИЯ-ПРЕДИКАТА ::=
    ИМЯ-ПРЕДИКАТА ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА

Имя предиката обозначает предикат, представленный описанием спецификации предиката.

ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА ::=
    ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ФУНКЦИИ |
    ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ФУНКЦИИ ::=
    ЗАГОЛОВОК-ПРЕДИКАТА [**pre** ПРЕДУСЛОВИЕ][**post** ПОСТУСЛОВИЕ]
ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
    ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ]
                                      [ПОСТУСЛОВИЯ-ВЕТВЕЙ]

6.3.3. Вызов предиката

Элементарным оператором программы является вызов предиката.

ВЫЗОВ-ПРЕДИКАТА ::= ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ |
                    ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ ::=
    ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА **(** [АРГУМЕНТЫ] **:** РЕЗУЛЬТАТЫ **)**
ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА ::= [ИМЯ-МОДУЛЯ **.**] ИМЯ-ПРЕДИКАТА |
                            ВЫРАЖЕНИЕ |
                            [ОБЪЕКТ-КЛАССА **.**] ИМЯ-ПРЕДИКАТА
ОБЪЕКТ-КЛАССА ::= ПЕРЕМЕННАЯ

Выражение должно быть предикатного типа. Значением выражения является предикат, запускаемый на исполнение данным вызовом.

АРГУМЕНТЫ ::= СПИСОК-ВЫРАЖЕНИЙ
СПИСОК-ВЫРАЖЕНИЙ ::= ВЫРАЖЕНИЕ [**,** СПИСОК-ВЫРАЖЕНИЙ]

Типы аргументов вызова должны быть совместимы (см. разд. 6.7) с типами соответствующих аргументов определения (или спецификации) вызываемого предиката.

РЕЗУЛЬТАТЫ ::= ПЕРЕМЕННАЯ [, РЕЗУЛЬТАТЫ] |
РЕЗУЛЬТАТ-ЛОКАЛ [, РЕЗУЛЬТАТЫ]
РЕЗУЛЬТАТ-ЛОКАЛ ::= ОПИСАНИЕ-ПЕРЕМЕННОЙ
ОПИСАНИЕ-ПЕРЕМЕННОЙ ::= ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ ИДЕНТИФИКАТОР
ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ ::=
ИЗОБРАЖЕНИЕ-ТИПА [:blank:] | **var [:blank:]**

Использование **var** возможно вместо соответствующего ИЗОБРАЖЕНИЯ-ТИПА, поскольку тип результата-локала в большинстве случаев можно восстановить по определению вызываемого предиката. Описатель **var** также может использоваться при описании локальной переменной в случае, когда тип этой переменной легко определяется из контекста дальнейшего использования переменной (см. разд. 6.4, 6.5).

Типы результатов вызова должны совпадать с типами соответствующих результатов в определении вызываемого предиката. Переменная, представленная описанием результата-локала, определяется как локальная в теле предиката, содержащем данный вызов. Описание локальной результирующей переменной внутри вызова эквивалентно описанию этой переменной перед вызовом.

Использование специального имени "_" в качестве результирующей переменной обозначает пустой результат: результат по этой позиции, получаемый при завершении исполнения вызова предиката, не используется в дальнейшем исполнении. Имя "_" зарезервировано в языке P – его нельзя использовать для именования объектов программы.

ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА **(** [АРГУМЕНТЫ] (**:** РЕЗУЛЬТАТЫ-ВЕТВИ)+ **)**
РЕЗУЛЬТАТЫ-ВЕТВИ ::= [РЕЗУЛЬТАТЫ] [ОПЕРАТОР-ПЕРЕХОДА]
ОПЕРАТОР-ПЕРЕХОДА ::= **#** МЕТКА

В качестве метки в операторе указывается либо метка ветви предиката, в теле которого находится данный вызов, либо метка ветви обработчика; подробнее см. в разд. 6.5.

Исполнение вызова предиката реализуется следующим образом. Результатом исполнения аргументов вызова является набор значений. Вычисление каждого аргумента вызова реализуется независимо от вычисления других, т. е. параллельно. Полученный набор значений аргументов присваивается соответствующим входным параметрам определения вызываемого предиката. Далее исполняется тело определения вызываемого предиката. В процессе исполнения тела определяется итоговая ветвь предиката и вычисляются значения результирующих переменных по этой ветви. Наконец, полученные значения результирующих переменных присваиваются соответствующим результирующим переменным вызова предиката, и исполнение вызова завершается по этой ветви вызова. Если в данной ветви вызова указан оператор перехода, то либо происходит переход на локальную метку, либо завершается тело предиката, содержащего вызов, той ветвью, которая указана в операторе перехода. Отметим, что подстановка аргументов и результатов предиката реализуется по значению.

ВЫЗОВ-ФУНКЦИИ ::= ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА **(** [АРГУМЕНТЫ] **)**

Вызов предиката-функции может иметь вид вызова функции. Результатом исполнения является набор значений результирующих переменных предиката.

```
real r;
sqrt (2 : r);
real q = sqrt (3);
Comp (list(int) s: : int d, list (int) r) // спецификация гиперфункции Comp
pre  1: s = nil
post 2: s = cons(d, r) ;
/* если список s пуст, реализуется первая ветвь гиперфункции, иначе – реализуется
   вторая ветвь с результатами: d – первый элемент, r – хвост списка */
elemTwo (list(int) s: int e #value : #empty)
pre  empty: s = nil or s.cdr = nil
{  Comp (s  : #empty : int e1, list(int) s1 #ok)
case ok: Comp (s1 : #empty : e, list(int) s2 #value);
}
post value: e = (s.cdr).car;
/* гиперфункция elemTwo извлекает второй элемент списка, если элемент сущест-
   вует  */
```

Пример 2. Вызовы и определения предикатов и гиперфункций, спецификация гиперфункции

В определении предиката elemTwo переход по локальной метке ok является излишним. Кроме того, результаты e1 и s2 далее нигде не используются. Определение предиката elemTwo, представленное ниже, исправляет отмеченные недостатки:

```
elemTwo (list(int) s: int e #value : #empty)
pre 1: s = nil or s.cdr = nil
{    Comp (s  : #empty : _, list(int) s1 );
     Comp (s1 : #empty : e, _ #value);
}
post value: e = s.cdr.car;
```

6.4. Программа

Программа состоит из одного или нескольких модулей. Модуль определяет независимую часть программы.

ОПИСАНИЕ-МОДУЛЯ ::= [ЗАГОЛОВОК-МОДУЛЯ] СПИСОК-ОПИСАНИЙ
ЗАГОЛОВОК-МОДУЛЯ ::= **module** ИМЯ-МОДУЛЯ [(ОПИСАНИЯ-АРГУМЕНТОВ)];
ИМЯ-МОДУЛЯ ::= ИДЕНТИФИКАТОР

Аргументы, описанные в круглых скобках, являются формальными параметрами модуля. Структура ОПИСАНИЙ-АРГУМЕНТОВ определена в разд. 6.3.1.

СПИСОК-ОПИСАНИЙ ::= ОПИСАНИЕ (; ОПИСАНИЕ)*
ОПИСАНИЕ ::=
    ИМПОРТ-МОДУЛЯ | ОПИСАНИЕ-ТИПА |
    ОПИСАНИЕ-ГЛОБАЛЬНЫХ-ПЕРЕМЕННЫХ |
    ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА | СПЕЦИФИКАЦИЯ-ПРЕДИКАТА |
    ОПИСАНИЕ-ФОРМУЛЫ | ОПРЕДЕЛЕНИЕ-КЛАССА |
    ОПИСАНИЕ-СООБЩЕНИЯ | ОПРЕДЕЛЕНИЕ-ПРОЦЕССА

ОПИСАНИЕ-ФОРМУЛЫ (см. разд. 6.9) определяет функцию или предикат; они встречаются в предусловиях, постусловиях и формулах. Три последние альтернативы ОПИСАНИЯ используются для задания процессов; см. разд. 6.10.

ИМПОРТ-МОДУЛЯ ::=

import ИМЯ-МОДУЛЯ [(АРГУМЕНТЫ)] [as ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ]

ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ ::= ИДЕНТИФИКАТОР

Имена модулей, встречающиеся в описаниях данного модуля, должны быть определены конструкцией ИМПОРТ-МОДУЛЯ. Если импортируемый модуль имеет формальные параметры, то в круглых скобках задается соответствующий набор фактических параметров. Конструкция АРГУМЕНТЫ определена в разд. 6.3.3. Правила подстановки фактических параметров на место формальных те же самые, что и для вызова предиката. ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ используется в теле модуля как эквивалент ИМЯ-МОДУЛЯ (АРГУМЕНТЫ).

ОПИСАНИЕ-ГЛОБАЛЬНЫХ-ПЕРЕМЕННЫХ ::=

ОПИСАНИЕ-ПЕРЕМЕННЫХ | ОПИСАНИЕ-ПЕРЕМЕННЫХ-С-ИНИЦИАЛИЗАЦИЕЙ

Глобальные переменные обычно определяют внешние параметры задачи. В нижеследующих определениях предикатов эти переменные считаются аргументами. Они используются в теле предиката, но не упоминаются в заголовке.

ОПИСАНИЕ-ПЕРЕМЕННЫХ ::=

ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ ИМЯ-ПЕРЕМЕННОЙ (, ИМЯ-ПЕРЕМЕННОЙ)*

ОПИСАНИЕ-ПЕРЕМЕННЫХ-С-ИНИЦИАЛИЗАЦИЕЙ ::=

ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ
(, ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ)*

ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ ::= ИМЯ-ПЕРЕМЕННОЙ = ВЫРАЖЕНИЕ |
ИМЯ-ПЕРЕМЕННОЙ

Тип выражения должен быть совместим с типом переменой, которой присваивается значение переменной.

Модуль определяет совокупность имен и обозначаемых ими объектов: предикатов, типов, переменных, полей структур, конструкторов и распознавателей объединений, процессов, классов и сообщений. Объекты, представленные описаниями модуля, определяют область глобальных имен. Каждое описание предиката определяет автономную систему локализации имен: параметров, меток и локальных переменных. В описании предиката одно имя не может использоваться для двух разных объектов. Если глобальное имя используется в определении предиката, то это имя не может определяться в качестве имени метки, локальной переменной или параметра.

Следующей областью локализации являются конструкции: изображение подтипа, определение массива, элемент агрегата вида “итерация выражений”. Переменные, определенные в каждой из этих конструкций, локализованы в конструкции и недоступны вне нее. Класс также определяет независимую область локализации. Поля класса доступны префиксацией объекта класса (см. разд. 6.6).

Исполнение программы начинается исполнением предиката с именем main. Рекомендован предикат со следующими параметрами: main(seq(string)argv: int ret_code). Параметр argv поставляет массив входных аргументов при исполнении программы. Параметру ret_code присваивается значение кода выхода.

Для вывода информации используется оператор print, аргументы которого – выражения и массивы.

6.5. Операторы

БЛОК ::= { ОПЕРАТОР }

ОПЕРАТОР ::= [ОПИСАНИЕ-ПЕРЕМЕННЫХ ;]{ ЗАМКНУТЫЙ-ОПЕРАТОР |
[ОПИСАНИЕ-ПЕРЕМЕННЫХ ;] ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР |
[ОПИСАНИЕ-ПЕРЕМЕННЫХ ;] ОПЕРАТОР-СУПЕРПОЗИЦИИ

Описанные переменные считаются локальными в теле предиката. Тип локальной переменной может быть также определен описателем **var** (см. разд. 6.3.3) в случае, когда тип легко определяется из контекста дальнейшего использования переменной.

ЗАМКНУТЫЙ-ОПЕРАТОР ::=
ОПЕРАТОР-ПРИСВАИВАНИЯ | ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ |
ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ] |
УСЛОВНЫЙ-ОПЕРАТОР | БЛОК [ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ] |
ОПЕРАТОР-ВЫБОРА | ОПРЕДЕЛЕНИЕ-ЛОКАЛЬНОГО-ПРЕДИКАТА

ОПЕРАТОР-ПРИСВАИВАНИЯ ::= ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ

В результате исполнения оператора присваивания переменной присваивается значение выражения. Типы выражения и переменной должны быть совместимы (см. разд. 6.7).

УСЛОВНЫЙ-ОПЕРАТОР ::=
ОПЕРАТОР-ЕСЛИ **else** ЗАМКНУТЫЙ-ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]

ОПЕРАТОР-ЕСЛИ ::= **if (** ВЫРАЖЕНИЕ **)** ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]

В операторе перехода указывается либо метка ветви гиперфункции, либо метка ветви обработчика, следующего за блоком, содержащим данный условный оператор.

```
if (x > 0)      s = 1
else    if (x < 0) s = -1
        else  s = 0
```

Пример 3. Использование условного оператора

ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР ::=
ЗАМКНУТЫЙ-ОПЕРАТОР (**| |** ЗАМКНУТЫЙ-ОПЕРАТОР)+

Операторы, образующие параллельный оператор, должны иметь непересекающиеся наборы результирующих переменных. Исполнение параллельного оператора слагается из исполнения входящих в него операторов. Операторы выполняются независимо друг от друга; они могут исполняться параллельно.

ОПЕРАТОР-СУПЕРПОЗИЦИИ ::= ОПЕРАТОР (**;** ОПЕРАТОР)+

В цепочке операторов, составляющих оператор суперпозиции, каждый следующий оператор использует значения локальных переменных, присвоенных предыдущими операторами. Каждая пара соседних операторов в цепочке должна быть связана хотя бы одной локальной переменной; если такой связи нет, их композиция должна быть оформлена параллельным оператором.

ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА ::= **switch (** ВЫРАЖЕНИЕ **)**

ОПЕРАТОР-ВЫБОРА ::= ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА
{ ОПЕРАТОР-АЛЬТЕРНАТИВЫ+
[**default :** ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]]
}

ОПЕРАТОР-АЛЬТЕРНАТИВЫ ::=
case АЛЬТЕРНАТИВА (, АЛЬТЕРНАТИВА)* **:** ОПЕРАТОР [
ОПЕРАТОР-ПЕРЕХОДА]

АЛЬТЕРНАТИВА ::= ВЫРАЖЕНИЕ | ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА

Выполняется тот оператор альтернативы, для которого значение АЛЬТЕРНАТИВЫ совпадает со значением выражения в заголовке оператора выбора. Оператор после default выполняется в случае, когда нет ОПЕРАТОРА-АЛЬТЕРНАТИВЫ с требуемым значением АЛЬТЕРНАТИВЫ. Трактовка операторов перехода та же, что и для условного оператора.

Если в позиции выражения в заголовке оператора выбора находится переменная типа объединения, то в качестве АЛЬТЕРНАТИВЫ указывается ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА. Особенности выполнения ОПЕРАТОРА-АЛЬТЕРНАТИВЫ определены в разд. 6.7.

ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ ::=
(**case** [МЕТКА-ВЕТВИ-ОБРАБОТЧИКА **:**] ОПЕРАТОР)+
МЕТКА-ВЕТВИ-ОБРАБОТЧИКА ::= МЕТКА

Оператор обработки ветвей следует за вызовом гиперфункции либо за блоком, содержащем операторы перехода. Если предыдущий оператор нормально завершается, то оператор обработки ветвей игнорируется. Нормальное завершение возможно и для вызова гиперфункции, когда на одной из ветвей гиперфункции опущена метка перехода, что означает суперпозицию (или параллельную композицию) с последующим оператором, а на другой ветви имеется переход на метку ветви обработчика или на метку ветви гиперфункции, в теле которой находится данный оператор.

Исполнение оператора обработки ветвей инициируется оператором перехода, находящимся в предыдущем операторе. При этом исполняется оператор, помеченный соответствующей меткой, после чего исполнение оператора обработки ветвей и предыдущего оператора считается завершившимся.

В вызове гиперфункции могут быть опущены переходы, а в последующем операторе обработки ветвей – метки ветвей обработчика. В этом случае порядок ветвей обработчика соответствует порядку ветвей вызова гиперфункции.

```
A(i: y #1: #2)
        case 1: B(y, i: u)
        case 2: D(i: u);
```

Пример 4. Вызов гиперфункции с обработчиком ветвей

ОПРЕДЕЛЕНИЕ-ЛОКАЛЬНОГО-ПРЕДИКАТА ::= ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА

Определение локального предиката допускает использование параметров и локальных переменных предиката, в теле которого это определение находится. Локальный предикат доступен лишь в теле предиката, внутри которого он описан.

6.6. Выражения

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ ::=
КОНСТАНТА | ПЕРЕМЕННАЯ | АГРЕГАТ | ИМЯ-ПРЕДИКАТА |
ГЕНЕРАТОР-ПРЕДИКАТА | ВЫЗОВ-ФУНКЦИИ | **(** ВЫРАЖЕНИЕ **)** | ИМЯ-ТИПА |
МОДИФИКАЦИЯ | ЭЛЕМЕНТ-СПИСКА | ОПРЕДЕЛЕНИЕ-МАССИВА |
КОНСТРУКТОР | РАСПОЗНАВАТЕЛЬ | ПОЛЕ-ОБЪЕДИНЕНИЯ

Имя типа может быть аргументом вызова предиката и изображения типа. Определение ЭЛЕМЕНТА-СПИСКА дается в разд. 6.7. Операция ОПРЕДЕЛЕНИЕ-МАССИВА описывается в разд. 6.8.3. Последние три альтернативы в определении ПЕРВИЧНОГО-ВЫРАЖЕНИЯ относятся к объектам типа объединения и описаны в разд. 6.7.

ПЕРЕМЕННАЯ ::= ИМЯ-ПЕРЕМЕННОЙ | ЭЛЕМЕНТ-МАССИВА | ПОЛЕ-СТРУКТУРЫ |
ПЕРЕМЕННАЯ-КЛАССА | ВЫРЕЗКА-МАССИВА |
МУЛЬТИПЕРЕМЕННАЯ

ВЫРЕЗКА-МАССИВА определена в разд. 6.8.

ИМЯ-ПЕРЕМЕННОЙ ::= ИДЕНТИФИКАТОР [**'**]

Переменная вида имя′ используется для именования результата предиката. Переменная имя′ склеивается с переменной имя, являющейся аргументом предиката (см. разд. 6.3.1).

ЭЛЕМЕНТ-МАССИВА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **[** ИНДЕКСЫ-МАССИВА **]**
ИНДЕКСЫ-МАССИВА ::= СПИСОК-ВЫРАЖЕНИЙ

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “массив”. Значения ИНДЕКСОВ-МАССИВА должны принадлежать типам индексов соответствующего типа массива.

ПОЛЕ-ПЕРЕМЕННОЙ ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **.** ИМЯ-ПОЛЯ

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “структура”.

ИМЯ-ПОЛЯ ::= ИДЕНТИФИКАТОР

ПЕРЕМЕННАЯ-КЛАССА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **.** ПЕРЕМЕННАЯ

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “класс”.

МУЛЬТИПЕРЕМЕННАЯ ::= **|** СПИСОК-ПЕРЕМЕННЫХ **|** |
СПИСОК-ПЕРЕМЕННЫХ

Два разных способа представления мультипеременной считаются равноценными.

СПИСОК-ПЕРЕМЕННЫХ ::= ПЕРЕМЕННАЯ [**,** СПИСОК-ПЕРЕМЕННЫХ]

Мультипеременная определяет упорядоченный набор переменных. Число переменных в наборе должно быть больше 1.

АГРЕГАТ ::= АГРЕГАТ-СТРУКТУРА | АГРЕГАТ-МАССИВ |
АГРЕГАТ-МНОЖЕСТВО | АГРЕГАТ-СПИСОК

Агрегат определяет значение структурного типа. Агрегат-массив определяет массив, агрегат-множество – подмножество некоторого множества – значение типа **set** (см. разд. 6.7), агрегат-структура – структуру (как набор полей), агрегат-список – список, заданный последовательностью элементов.

АГРЕГАТ-СТРУКТУРА ::= [ИЗОБРАЖЕНИЕ-ТИПА] **(** [ЭЛЕМЕНТЫ-АГРЕГАТА] **)**
АГРЕГАТ-МАССИВ ::= [ИЗОБРАЖЕНИЕ-ТИПА] **[** [ЭЛЕМЕНТЫ-АГРЕГАТА] **]**
АГРЕГАТ-МНОЖЕСТВО ::= [ИЗОБРАЖЕНИЕ-ТИПА] **{** [ЭЛЕМЕНТЫ-АГРЕГАТА] **}**
АГРЕГАТ-СПИСОК ::= [ИЗОБРАЖЕНИЕ-ТИПА] **[[** ЭЛЕМЕНТЫ-АГРЕГАТА **]]**

Агрегат [ ] обозначает пустой массив (когда верхняя граница индексов меньше нижней). Агрегат { } определяет пустое множество. Если пара скобок [[или]] встречается рядом при изображении АГРЕГАТ-МАССИВА, то соседние скобки дожны быть разделены пробелом.

ЭЛЕМЕНТЫ-АГРЕГАТА ::= ЭЛЕМЕНТ-АГРЕГАТА [**,** ЭЛЕМЕНТЫ-АГРЕГАТА]

Значением агрегата является набор значений элементов агрегата, перечисленных в скобках. Агрегат может иметь иерархическую структуру, поскольку элементом агрегата может быть агрегат. Тип агрегата определяется типом позиции, в которой находится агрегат. Значение агрегата должно соответствовать типу позиции. Тип агрегата может быть указан явно ИЗОБРАЖЕНИЕМ-ТИПА – это не является необходимым, но может улучшить восприятие программы.

ЭЛЕМЕНТ-АГРЕГАТА ::= [ИНДЕКС-ЭЛЕМЕНТА-АГРЕГАТА **:**] ВЫРАЖЕНИЕ
ИНДЕКС-ЭЛЕМЕНТА-АГРЕГАТА ::= ВЫРАЖЕНИЕ | ИМЯ-ПОЛЯ

Индексы элементов, если присутствуют, должны соответствовать индексам элементов массива или именам полей структуры.

```
type Ar4 = array (real, 1..4);
Ar4 x = [ 0, 1, 2, 3];
```

Пример 5. Задание агрегата

ГЕНЕРАТОР-ПРЕДИКАТА ::=
predicate ОПИСАНИЕ-ПРЕДИКАТА

Исполнение генератора предиката создает новый предикат, операторная часть которого определяется телом предиката, находящегося в составе ОПИСАНИЯ-ПРЕДИКАТА; см. разд. 6.3.1. В теле предиката фиксируются значения свободных переменных (отличных от аргументов и результатов) на момент исполнения генератора. Новый предикат становится значением генератора предиката.

УНАРНОЕ-ВЫРАЖЕНИЕ ::= УНАРНАЯ-ОПЕРАЦИЯ ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ
УНАРНАЯ-ОПЕРАЦИЯ ::= **+** | **-** | **!** | **~**
БИНАРНОЕ-ВЫРАЖЕНИЕ ::= ВЫРАЖЕНИЕ БИНАРНАЯ-ОПЕРАЦИЯ ВЫРАЖЕНИЕ
БИНАРНАЯ-ОПЕРАЦИЯ ::= ***** | **/** | **%** | **+** | **-** | **<<** | **>>** | **in** |
< | **>** | **<=** | **>=** | **=** | **!=** | **&** | **^** | **or** | **xor** | **=>** | **<=>**

Семантика операций и их приоритеты определены ниже в таблице. Типы аргументов каждой операции должны соответствовать операции. Операция “+” для объединения массивов описана в разд. 6.8.4.

МОДИФИКАЦИЯ ::= МОДИФИКАЦИЯ-СТРУКТУРЫ | МОДИФИКАЦИЯ-МАССИВА
МОДИФИКАЦИЯ-СТРУКТУРЫ ::= ВЫРАЖЕНИЕ **with** АГРЕГАТ
МОДИФИКАЦИЯ-МАССИВА ::=
ВЫРАЖЕНИЕ **with** АГРЕГАТ |
ВЫРАЖЕНИЕ **with** ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ

В структурном значении выражения, обозначающего массив или структуру, меняются компоненты, представленные АГРЕГАТОМ или ОПРЕДЕЛЕНИЕМ-МАССИВА-ПО-ЧАСТЯМ (см. разд. 6.8.3). Значением операции является обновленный массив или структура.

```
type ar0_4 = array (int, 0..4);
ar0_4 ones = [ 1, 1, 1, 2, 1 ];
ar0_4 ones1 = ones [3: 1]; // [ 1, 1, 1, 1, 1 ]
type Vec(nat n) = array (real, 1..n);
perm(nat n, Vec(n) b, nat m, k : Vec(n) b')
{ b' = b with [k: b [m], m: b [k]] };
```

Пример 6. Замена части структурного значения

УСЛОВНОЕ-ВЫРАЖЕНИЕ ::= ВЫРАЖЕНИЕ **?** ВЫРАЖЕНИЕ **:** ВЫРАЖЕНИЕ

Первое выражение имеет тип “логический”. Если первое ВЫРАЖЕНИЕ завершается илентификатором, то идентификатор и последующий символ “?” должны быть разделены пробелом.

МУЛЬТИВЫРАЖЕНИЕ ::= **|** СПИСОК-ВЫРАЖЕНИЙ **|** |
СПИСОК-ВЫРАЖЕНИЙ

Значением мультивыражения является набор значений, получающихся в результате вычисления перечисленных выражений. Выражения вычисляются независимо, т. е. параллельно. Мультивыражение может находиться в правой части оператора присваивания, если в левой находится мультипеременная.

ВЫРАЖЕНИЕ ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ | УНАРНОЕ-ВЫРАЖЕНИЕ |
БИНАРНОЕ-ВЫРАЖЕНИЕ | УСЛОВНОЕ-ВЫРАЖЕНИЕ |
МУЛЬТИВЫРАЖЕНИЕ

Таблица 1. Операции в порядке уменьшения приоритета

^	Возведение в степень
+, -, !, ~	Унарные плюс и минус, логическое отрицание и поэлементное дополнение для множеств, побитовое дополнение для ограниченных целых
***, /, %**	Арифметическое умножение, деление и остаток от целочисленного деления
+, -	Арифметическое сложение и вычитание; объединение и вычитание множеств; конкатенация строк; объединение массивов с непересекающимися типами индексов
<<, >>	Побитовый (поэлементный) сдвиг влево и вправо для целых
<-	Обновление элемента массива или поля структуры
in	Проверка вхождения элемента в множество
<, >, <=, >=	Операции арифметического сравнения
=, !=	Проверка на равенство и неравенство
&	Пересечение множеств, логическая конъюнкция, побитовое И для ограниченных целых
xor	Симметрическая разность для множеств, исключительное ИЛИ для логических выражений и ограниченных целых (побитовое)
or	Объединение множеств, логическая дизъюнкция, побитовое ИЛИ для ограниченных целых
=>	Импликация
<=>	Логическое тождество
?:	Трёхместная условная операция

6.7. Типы

Всякая переменная имеет некоторый тип. Тип может быть атрибутом языковой конструкции. Семантика конструкции налагает определенные ограничения на значения типов. Для каждой подконструкции некоторой конструкции определяется *позиция* ее вхождения внутри конструкции. Это, например, позиции операндов операций, позиции правой и левой частей в операторе присваивания, позиция фактического параметра в вызове предиката и др. При наличии в конструкции двух позиций типы этих позиций должны быть *согласованы*. Частным случаем согласованности является *совместимость* типов. Например, тип выражения правой части оператора присваивания должен быть совместим с типом переменной левой части оператора присваивания, но не наоборот. Реализуется неявное преобразование (называемое *приведением*) значения совместимого типа к типу, требуемому позицией.

ОПИСАНИЕ-ТИПА ::=
type ИМЯ-ТИПА [**(** ПАРАМЕТРЫ-ТИПА **)**] = ИЗОБРАЖЕНИЕ-ТИПА |
ПРЕДОПИСАНИЕ-ТИПА
ИМЯ-ТИПА ::= ИДЕНТИФИКАТОР
ПАРАМЕТРЫ-ТИПА ::=
ИЗОБРАЖЕНИЕ-ТИПА [:blank:] ИДЕНТИФИКАТОР (**,** ИДЕНТИФИКАТОР)*
[**,** ПАРАМЕТРЫ-ТИПА]

Описание типа связывает имя типа с его изображением. Тип может быть параметризован, т. е. зависеть от набора значений переменных и типов, указанных в качестве параметров.

ПРЕДОПИСАНИЕ-ТИПА ::= **type** ИМЯ-ТИПА

Если имя типа используется до его описания, что возможно для рекурсивных определений типов, то перед первым вхождением имени оно должно быть декларировано с помощью предописания.

ИЗОБРАЖЕНИЕ-ТИПА ::=
ИЗОБРАЖЕНИЕ-ПРИМИТИВНОГО-ТИПА | ИЗОБРАЖЕНИЕ-ТИПА-ПО-ИМЕНИ |
ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА | ИЗОБРАЖЕНИЕ-ТИПА-ПРЕДИКАТА |
ИЗОБРАЖЕНИЕ-ПОДТИПА | ИЗОБРАЖЕНИЕ-ПЕРЕЧИСЛЕНИЯ |
ИЗОБРАЖЕНИЕ-СТРУКТУРНОГО-ТИПА |
ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ
ИЗОБРАЖЕНИЕ-ПРИМИТИВНОГО-ТИПА ::=
nat [**(**РАЗРЯДНОСТЬ-ЦЕЛЫХ**)**] |
int [**(**РАЗРЯДНОСТЬ-ЦЕЛЫХ**)**] |
real [**(**РАЗРЯДНОСТЬ-ВЕЩЕСТВЕННЫХ**)**] |
bool | **char**

Тип **nat** является подмножеством неотрицательных чисел типа **int**, **char** – множество символов некоторого алфавита. Конкретные представления примитивных типов даются в императивном расширении языка (см. разд. 6.11).

Разрядность определяет максимальное число битов в представлении значений типа. Указание разрядности в изображении примитивного типа является частью императивного расширения языка.

Значения типа **nat** совместимы с типом **int**, а **int** – с типом **real**. Значения типа меньшей размерности совместимы с аналогичными типами большей размерности.

ИЗОБРАЖЕНИЕ-ТИПА-ПО-ИМЕНИ ::= [ИМЯ-МОДУЛЯ **.**] ИМЯ-ТИПА [**(** АРГУМЕНТЫ **)**]

Ранее определенный тип может быть представлен своим именем. Изображение параметризованного типа представляется с конкретными параметрами.

ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА ::= **type** ИМЯ-ТИПА

Объявление типа в качестве параметра описания типа или определения предиката реализуется описателем **type**. В качестве обозначения типа может использоваться выражение типа **type**, которым, например, может являться параметр предиката или параметризованного типа. Например:

```
type Point (type T) = struct (T x, y);
type Polyline (type T) = list (Point (T));
decimate (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?
           nil :
           data.car + (len(data) < 3 ? nil : decimate(T, data.cdr.cdr))
};
reverse (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?  nil :  reverse (T, data.cdr) + data.car
};
Polyline (real) line = [[ (0.0, 0.0), (0.1, 0.2), (0.2, 0.5),
                          (0.3, 0.6), (0.4, 0.3), (0.5, 0.0) ]];
Polyline (real) line1 = reverse (decimate (real, line));
// [[ (0.4, 0.3), (0.2, 0.5), (0.0, 0.0) ]]
```

Пример 7. Использование типов в качестве параметров

В примере выше идентификатор T использован в качестве типа-параметра, а тип **real** – как фактический параметр.

```
squareVec (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?  nil :  data.car ^ 2 + squareVec (T, data.cdr)
};
list (int) squares = squareVec (int, [[ 0, 1, 2, 3, 4 ]] ); // [[ 0, 1, 4, 9, 16 ]]
```

Выше приведен еще один пример использования типа T в качестве параметра предиката.

ИЗОБРАЖЕНИЕ-ТИПА-ПРЕДИКАТА ::=
predicate ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА

Имя параметра предиката может быть опущено. В этом случае описывается только тип параметра. Изображение типа предиката используется для описания переменной предикатного типа.

```
type int_list   = list (int);
map_list (int_list src, predicate (int : int) func : int_list dest) {
  dest =  (src = nil) ?  nil :  func (src.car) + map_list (src.cdr, func))
};
int_list numbers = [[ 0, 1, 2, 3, 4, 5 ]];
int_list squares = map_list (numbers, predicate (int a : int b) { b = a * a; });
    // squares = [[ 0, 1, 4, 9, 16, 25 ]]
```

Пример 8. Использование предикатного типа

ОПРЕДЕЛЕНИЕ-ПОДТИПА ::=
subtype (ОПИСАНИЕ-ПЕРЕМЕННОЙ : ВЫРАЖЕНИЕ **)** |
ИЗОБРАЖЕНИЕ-ДИАПАЗОНА

Конструкция **subtype**(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 подтипом типа T. Если выражение содержит другие переменные, то все они являются параметрами изображаемого типа. Конструкция ОПИСАНИЕ-ПЕРЕМЕННОЙ определена в разд. 6.3.3

ИЗОБРАЖЕНИЕ-ДИАПАЗОНА ::= ВЫРАЖЕНИЕ **..** ВЫРАЖЕНИЕ

Изображение диапазона является подмножеством типа **int**, **enum** или **char**. Изображение типа **subtype**(**nat** i: i >= 1 & i <= n + 1) эквивалентно изображению диапазона: 1 .. n + 1.

```
type odd_t = subtype (int n: n % 2 = 1);
type diap10_20_t = 10..20;
```

ИЗОБРАЖЕНИЕ-СТРУКТУРНОГО-ТИПА ::=
ИЗОБРАЖЕНИЕ-ТИПА-СТРУКТУРЫ | ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ |
ИЗОБРАЖЕНИЕ-ТИПА- СПИСКА | ИЗОБРАЖЕНИЕ-ТИПА-МНОЖЕСТВА |
ИЗОБРАЖЕНИЕ-ТИПА-МАССИВА

ИЗОБРАЖЕНИЕ-ТИПА-СТРУКТУРЫ ::= **struct (** ОПИСАНИЕ-ПОЛЕЙ **)**

ОПИСАНИЕ-ПОЛЕЙ ::= ИЗОБРАЖЕНИЕ-ТИПА [:blank:] СПИСОК-ИМЕН-ПОЛЕЙ
[**,** ОПИСАНИЕ-ПОЛЕЙ]

СПИСОК-ИМЕН-ПОЛЕЙ ::= ИМЯ-ПОЛЯ [**,** СПИСОК-ИМЕН-ПОЛЕЙ]

Значение типа структуры состоит из совокупности значений полей структуры. Имена полей должны быть различны. Число полей должно быть не менее двух.

```
type Point = struct ( int x, y );
Point pt = (10, 20);
```

Пример 9. Определение структуры

ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ ::= **union (** ОПИСАНИЯ-КОНСТРУКТОРОВ **)**

ОПИСАНИЯ-КОНСТРУКТОРОВ ::=
ОПИСАНИЕ-КОНСТРУКТОРА (**,** ОПИСАНИЕ-КОНСТРУКТОРА)*

Значением типа объединения является значение одного из конструкторов, перечисленных в списке описаний конструкторов. Число конструкторов должно быть не меньше двух.

ОПИСАНИЕ-КОНСТРУКТОРА ::= ИМЯ-КОНСТРУКТОРА [**(** ОПИСАНИЕ-ПОЛЕЙ **)**]

ИМЯ-КОНСТРУКТОРА ::= ИДЕНТИФИКАТОР

Поля, определяемые в круглых скобках, являются полями объединения. Значения этих полей используются в качестве аргументов конструктора. Имена полей объединения по всем конструкторам должны быть различны.

Работа с объектами типа объединения реализуется с помощью следующих конструкций: конструктор, распознаватель и поле объединения. Все они являются ПЕРВИЧНЫМИ-ВЫРАЖЕНИЯМИ (см. разд. 6.6).

КОНСТРУКТОР ::= ИМЯ-КОНСТРУКТОРА [(СПИСОК-ВЫРАЖЕНИЙ)]

Выражения, подставляемые в качестве аргументов КОНСТРУКТОРА, по их числу и типам должны соответствовать ОПИСАНИЯМ-ПОЛЕЙ в соответствующем ОПИСАНИИ-КОНСТРУКТОРА. Правила соответствия такие же, как для вызова предиката. Значением конструкции КОНСТРУКТОР является имя конструктора вместе со значениями его аргументов, если аргументы присутствуют.

РАСПОЗНАВАТЕЛЬ ::= ИМЯ-РАСПОЗНАВАТЕЛЯ **(** ВЫРАЖЕНИЕ **)**

ИМЯ-РАСПОЗНАВАТЕЛЯ ::= ИМЯ-КОНСТРУКТОРА **?**

Символ “**?**” является частью имени распознавателя и пишется слитно с именем конструктора, без пробела. ВЫРАЖЕНИЕ имеет тип объединения. Распознаватель является функцией типа **bool**. Значение РАСПОЗНАВАТЕЛЯ истинно, если значением ВЫРАЖЕНИЯ является конструктор с именем ИМЯ-КОНСТРУКТОРА.

ПОЛЕ-ОБЪЕДИНЕНИЯ ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **.** ИМЯ-ПОЛЯ

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа «объединение». Значением ПОЛЯ-ОБЪЕДИНЕНИЯ является значение поля с именем ИМЯ-ПОЛЯ в структуре переменной типа «объединение». Использование конструкции ПОЛЕ-ОБЪЕДИНЕНИЯ корректно лишь в случае истинности распознавателя ИМЯ-КОНСТРУКТОРА?(ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ) для конструктора, к которому относится поле с именем ИМЯ-ПОЛЯ, в противном случае исполнение конструкции не определено.

Имена конструкторов и распознавателей являются глобально определенными в теле модуля, содержащего ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ.

ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА ::=

ИМЯ-КОНСТРУКТОРА [**(** ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА **)**]

ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА ::=

ОПРЕДЕЛЕНИЕ-ПОЛЯ-КОНСТРУКТОРА

[**,** ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА]

ОПРЕДЕЛЕНИЕ-ПОЛЯ-КОНСТРУКТОРА ::=

[ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ] ИДЕНТИФИКАТОР

Конструкция ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА используется в качестве АЛЬТЕРНАТИВЫ после **case** в операторе выбора (см. разд. 6.5) для выражения типа объединения. Значением АЛЬТЕРНАТИВЫ является конструктор с именем ИМЯ-КОНСТРУКТОРА и набором переменных, обозначающих поля данного конструктора. Эти переменные являются локальными в ОПЕРАТОРЕ-АЛЬТЕРНАТИВЫ, исполняемом для данной АЛЬТЕРНАТИВЫ. Типы переменных могут не указываться, поскольку они определяются из ИЗОБРАЖЕНИЯ-ТИПА-ОБЪЕДИНЕНИЯ.

```
type int_tree = union (
   leaf (int val),
   node (int_tree lhs, int v, int_tree rhs)
);

int_tree foo;
// ...
switch (foo) {
   case leaf(v0): print("leaf", v0)
   case node(l, v1, r): print("node with ", v1)
};
int_tree one_leaf = leaf(42);
int_tree small_tree = node(leaf(1), 2, node(leaf(3), 4, leaf(5)));
// small_tree задаёт следующее дерево:
//    2
//   / \
//  1   4
//     / \
//    3   5
```

Пример 10. Использование объединений

ИЗОБРАЖЕНИЕ-ПЕРЕЧИСЛЕНИЯ ::=
enum (ИДЕНТИФИКАТОР (**,** ИДЕНТИФИКАТОР)* **)**

Тип перечисления является частным случаем типа объединения с конструкторами без аргументов. Описатель enum трактуется как эквивалент **union**.

```
type fruit = enum (apple, orange, banana);
```

Пример 11. Описание перечисления

Структура вида “список” представляется как упорядоченная совокупность однородных элементов. Тип списка из элементов типа T определяется следующим описанием:

```
type list (type T) = union (
     nil,
     cons (T car, list(T) cdr)
);
```

Тип list считается определенным в языке P и не требует описания в программе. Имена list, nil и cons глобально определены в предикатной прграмме.

Конструктор nil определяет пустой список, не содержащий элементов. Остальные значения типа list представляются конструктором cons(x, s), где элемент x есть первый элемент списка – голова списка, а s есть оставшаяся часть списка – хвост списка. Функция len(s) определяет число элементов списка s. Операция конкатенации s1 + s2 определяет список, состоящий из элементов списка s1, за которыми следуют элементы списка s2. Аргументами конкатенации могут быть также элементы списка. Функция last(s) определяет последний элемент непустого списка s; функция prec(s) – оставшуюся часть списка без последнего элемента.

ЭЛЕМЕНТ-СПИСКА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ [ИНДЕКС-ЭЛЕМЕНТА]

Данная конструкция определяет значение элемента с указанным индексом в списке. Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является список.

ИНДЕКС-ЭЛЕМЕНТА ::= ВЫРАЖЕНИЕ

Для списка s конструкция s[i] определяет i-й элемент. Допустимыми являются значения индекса i от 0 до len(s) – 1.

```
type int_list   = list (int);
int_list s      = [[ 1, 1, 2, 3, 5, 8 ]];
int s_car       = s.car;       // s_car = 1
int_list s_cdr  = s.cdr;       // s_cdr = [[ 1, 2, 3, 5, 8 ]]
int_list ss     = s + [[ 13, 21 ]];  // ss = [[ 1, 1, 2, 3, 5, 8, 13, 21 ]]
int count       = len (s);     // count = 6
int e           = s[3];        // e = 3
```

Пример 12. Операции со списком

СТРОКОВЫЙ-ТИП ::= **string**

Строковый тип **string** является обозначением типа списка list(char). В дополнение к определенным выше операциям со списками имеется возможность задания конкретного значения строкового типа в виде строковой константы (см. разд. 6.2).

ИЗОБРАЖЕНИЕ-ТИПА-МНОЖЕСТВА ::= **set (** ИЗОБРАЖЕНИЕ-БАЗОВОГО-ТИПА **)**
ИЗОБРАЖЕНИЕ-БАЗОВОГО-ТИПА ::= ИЗОБРАЖЕНИЕ-ТИПА

Тип множества есть множество всех подмножеств некоторого конечного базового типа. Имеется унарная операция ~ дополнения множества. Определены: операция «+» сложения множеств, операция «–» вычитания множеств, а также вычитания элемента из множества, операция & пересечения множеств, операция **or** объединения множеств, операция **in** принадлежности элемента множеству (см. разд. 6.6). Число элементов в множестве реализуется функцией len.

Операция mask преобразует множество в целое значение. Операция bits реализует обратную операцию преобразования целого в множество.

```
type int_set_t      = set (1..1000);
int_set_t pow_2     = { 0, 2, 4, 8, 16, 32 };
bool is_pow_2       = 4 in pow_2; // is_pow_2 = true
int_set_t more_pow_2 = pow_2 + { 64, 128 };
```

Пример 13. Использование множеств

6.8. Массивы

6.8.1. Описание типа массива

ИЗОБРАЖЕНИЕ-ТИПА-МАССИВА ::=
array (ИЗОБРАЖЕНИЕ-ТИПА-ЭЛЕМЕНТА **,** ИЗМЕРЕНИЯ-МАССИВА **)**
ИЗОБРАЖЕНИЕ-ТИПА-ЭЛЕМЕНТА ::= ИЗОБРАЖЕНИЕ-ТИПА
ИЗМЕРЕНИЯ-МАССИВА ::= ИЗОБРАЖЕНИЕ-ТИПА-ИНДЕКСА [**,** ИЗМЕРЕНИЯ-МАССИВА]
ИЗОБРАЖЕНИЕ-ТИПА-ИНДЕКСА ::= ИЗОБРАЖЕНИЕ-ТИПА

Значение массива состоит из совокупности элементов. Каждый элемент доступен по набору индексов в массиве: для массива элемент с набором индексов m есть A[m]. Число индексов совпадает с числом измерений массива. Тип каждого индекса определяет совокупность допустимых значений индекса и должен быть конечным.

Имеется операция «+» для объединения двух массивов-операндов в один массив при условии, что типы элементов совпадают, а множества наборов индексов операндов совместимы по типам и не пересекаются.

```
type int_vec = array (int, 1..5);
type int_mtx = array (int, 1..2, 1..3);
int_vec v = [ 3, 1, 4, 1, 5 ];
int_vec w = for (x) x*x;
int v_2 = v [2]; // v_2 = 1
int_mtx m = [ [ 1, 2, 3 ], [ 104, 5, 6 ] ];
int e = m [2, 3]; // e = 6
```

Пример 14. Описание массивов и их инициализация

6.8.2. Вырезка массива

ВЫРЕЗКА-МАССИВА ::=
ВЫРАЖЕНИЕ-МАССИВ **[** СУЖЕННЫЙ-НАБОР-ТИПОВ-ИНДЕКСОВ **]**
ВЫРАЖЕНИЕ-МАССИВ ::= ВЫРАЖЕНИЕ
СУЖЕННЫЙ-НАБОР-ТИПОВ-ИНДЕКСОВ ::= НАБОР-ИНДЕКСОВ
НАБОР-ИНДЕКСОВ ::= ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ [**,** НАБОР-ИНДЕКСОВ]
ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ ::= ЗНАЧЕНИЕ-ИНДЕКСА | ИЗОБРАЖЕНИЕ-ДИАПАЗОНА
ЗНАЧЕНИЕ-ИНДЕКСА ::= ВЫРАЖЕНИЕ

Вырезка массива определяет массив как часть некоторого массива заданием подмножества наборов индексов. Достаточно задать сужение хотя бы по одному измерению массива. В частности, допустимо сужение к единственному значению индекса. Однако вырезка массива не может превратиться в единственный элемент массива. Результатом исполнения вырезки массива является переменная типа “массив”. Нового массива при этом не создается.

6.8.3. Определение массива

ОПРЕДЕЛЕНИЕ-МАССИВА ::=
ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА |
АГРЕГАТ-МАССИВ |
ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ
ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ ::=
ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА

Результатом вычисления ОПРЕДЕЛЕНИЯ-МАССИВА является значение массива. Вычисление реализуется итерацией по всевозможным значениям набора индексов массива. Для каждого набора индексов вычисляется соответствующий элемент массива. Вычисление значений разных элементов может проводиться параллельно.

ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА ::= ВЫРАЖЕНИЕ

ВЫРАЖЕНИЕ для элемента массива зависит от индексов, указанных в ОПРЕДЕЛЕНИИ-ИНДЕКСОВ.

ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ ::= **for (** ЗАДАНИЕ-ИНДЕКСОВ **)**
ЗАДАНИЕ-ИНДЕКСОВ ::= ОПРЕДЕЛЕНИЕ-ИНДЕКСА [**,** ЗАДАНИЕ-ИНДЕКСОВ]
ОПРЕДЕЛЕНИЕ-ИНДЕКСА ::= [ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ] ИНДЕКС
ИНДЕКС ::= ИДЕНТИФИКАТОР

Переменные, обозначающие индексы, локальны в ОПРЕДЕЛЕНИИ-МАССИВА. При определении индекса обычно используется описатель **var** (см. разд. 6.4). Указание типа индекса требуется в редких случаях, когда тип масива (в том числе и типы индексов массива) трудно определить из позиции, в которой находится ОПРЕДЕЛЕНИЕ-МАССИВА. Отметим, что в ОПРЕДЕЛЕНИИ-ИНДЕКСА тип индекса, если он задан явно, должен точно покрывать множество значений индекса; обычно это диапазон типа **int**.

```
type ar1_5 = array (int, 1..5);
ar1_5 squ;
squ = for (var i) i*i;
ar1_5 r = for (var i) 100 - i;
```

Пример 15. Конструкторы массивов

ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА ::=
{ (ОПРЕДЕЛЕНИЕ-ЧАСТИ-МАССИВА)+
[**default :** ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА]
}
ОПРЕДЕЛЕНИЕ-ЧАСТИ-МАССИВА ::=
case ИНДЕКСЫ-ЧАСТИ **:** ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА
ИНДЕКСЫ-ЧАСТИ ::= ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ |
(НАБОР-ИНДЕКСОВ **)** |
(ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ [**,** ИНДЕКСЫ-ЧАСТИ] **)** |
((НАБОР-ИНДЕКСОВ **)** [**,** ИНДЕКСЫ-ЧАСТИ] **)**

ИНДЕКСЫ-ЧАСТИ определяют некоторое подмножество на произведении типов индексов. Эти подмножества не должны пересекаться для разных ОПРЕДЕЛЕНИЙ-ЧАСТИ-МАССИВА. Определение элементов массива для наборов индексов, не принадлежащих ни одной из указанных частей массива, реализуется частью **default**. При отсутствии части **default** объединение подмножеств наборов индексов для разных частей должно совпадать с полным множеством наборов индексов. Вычисление элементов массива по каждой из частей массива проводится независимо, возможно параллельно.

Если ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА используется в операции модификации массива (см. разд. 6.6), то часть **default** отсутствует.

```
type Ar(nat k) = array (real, 1..k);
F (nat n, Ar(n) x: Ar(n+1) x')
{ x' = for (var j) { case 1..n : x[j] + 1 case n + 1 : 0  } }
```

Пример 16. Определение массива по частям

```
type MATR(nat k) = array(real, 1..k, 1..k);
perm_lines(nat n, MATR(n) a, nat k, m : MATR(n) a' )
pre  1 <= k < m <= n
{       a' = a  with for (var i, j) {
            case (k, 1..n): a[m, j]
            case (m, 1..n): a[k, j]
          }
}
```

Пример 17. Перестановка двух строк матрицы

В матрице a переставляются местами строки с номерами k и m. Перестановка реализуется применением операции модификации (см. разд. 6.6) двух строк в массиве a; остальные строки остаются неизменными. Задание нового значения двух строк матрицы реализуется конструкцией ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА.

6.8.4. Объединение массивов

Операндами операции «+» объединения массивов:

ВЫРАЖЕНИЕ-МАССИВ **+** ВЫРАЖЕНИЕ-МАССИВ

являются выражения, значениями которых являются массивы. Объединяемые массивы должны иметь совпадающие типы элементов и непересекающиеся типы индексов, причем

типы индексов должны принадлежать некоторому общему типу. Результатом операции является массив. Тип индексов результирующего массива есть объединение типов индексов операндов, а тип элементов тот же, что и у операндов. Произвольный элемент результирующего массива есть либо элемент первого массива-операнда, либо элемент второго.

6.9. Формулы

Формулы, используемые в качестве предусловий и постусловий предикатов, есть формулы типизированного исчисления предикатов высших порядков. Подформула, входящая в предусловие или постусловие, может быть определена отдельно в виде описания формулы.

ОПИСАНИЕ-ФОРМУЛЫ ::=
formula ИМЯ-ФОРМУЛЫ **(** ОПИСАНИЯ-АРГУМЕНТОВ [: ТИП-РЕЗУЛЬТАТА] **) =**
ФОРМУЛА
ИМЯ-ФОРМУЛЫ ::= ИДЕНТИФИКАТОР
ТИП-РЕЗУЛЬТАТА ::= ИЗОБРАЖЕНИЕ-ТИПА

ОПИСАНИЕ-ФОРМУЛЫ вводит имя формулы для обозначения выражения, представленного ФОРМУЛОЙ. Переменные, входящие в ФОРМУЛУ, должны быть указаны в ОПИСАНИЯХ-АРГУМЕНТОВ (см. разд. 6.3.1). Описание формулы помещается среди описаний модуля программы (см. разд. 4). ФОРМУЛА является выражением типа ТИП-РЕЗУЛЬТАТА. Если ТИП-РЕЗУЛЬТАТА не указан, то ФОРМУЛА имеет тип **bool**.

Использование формулы, определенной ОПИСАНИЕМ-ФОРМУЛЫ, в других формулах реализуется через вызов формулы.

ВЫЗОВ-ФОРМУЛЫ ::= ИМЯ-ФОРМУЛЫ **(** СПИСОК-ВЫРАЖЕНИЙ **)**

Описание формулы может быть рекурсивным: ФОРМУЛА в правой части описания может содержать рекурсивный вызов этой формулы. Не допускается взаимной рекурсии в описаниях двух разных формул. Другое требование: рекурсивный вызов должен находиться в позитивной позиции ФОРМУЛЫ. Понятия позитивной и негативной позиции определены ниже.

ФОРМУЛА ::= ВЫРАЖЕНИЕ | **(** ФОРМУЛА **)** | ВЫЗОВ-ФОРМУЛЫ |
! ФОРМУЛА | ФОРМУЛА **&** ФОРМУЛА | ФОРМУЛА **or** ФОРМУЛА |
ФОРМУЛА **=>** ФОРМУЛА | ФОРМУЛА **<=>** ФОРМУЛА |
КВАНТОРНАЯ-ЧАСТЬ ФОРМУЛА

Все альтернативы приведенного правила, кроме первых трех, определяют формулы типа **bool**, т. е. предикаты. Операции «**!**», «**&**» и «**or**» имеют тот же смысл, что и для логических выражений. Операция => обозначает импликацию, <=> – логическое тождество.

КВАНТОРНАЯ-ЧАСТЬ ::=
КВАНТОР СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ **.** [КВАНТОРНАЯ-ЧАСТЬ]
КВАНТОР ::= КВАНТОР-ВСЕОБЩНОСТИ | КВАНТОР-СУЩЕСТВОВАНИЯ
КВАНТОР-ВСЕОБЩНОСТИ ::= **!** | **forall**
КВАНТОР-СУЩЕСТВОВАНИЯ ::= **?** | **exists**

Использование **forall** вместо «**!**» предпочтительно при вхождении квантора внутри формулы, поскольку «**!**» ассоциируется также с отрицанием.

СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ ::=
[ИЗОБРАЖЕНИЕ-ТИПА [:blank:]] ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ
(**,** ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ)*
ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ ::= ИДЕНТИФИКАТОР

Приоритеты логических связок в порядке убывания следующие: **!**, **&**, **=>**, **<=>**, кванторы **!**, **forall**, **?** и **exists**.

Определим понятие позиции, позитивной или негативной, для произвольного вхождения подформулы. Позиция ФОРМУЛЫ в качестве правой части ОПИСАНИЯ-ФОРМУЛЫ является позитивной. Допустим, формула X есть одна из следующих формул: A & B, A **or** B, !C,

C => A, !y.A, ?y.A. Для перечисленных случаев подформулы A и B имеют ту же позицию, что формула X, а подформула C меняет позицию на противоположную. Все остальные позиции подформул считаются неизвестными.

6.10. Процессы

Для большинства программ можно построить спецификацию в виде предусловия и постусловия. Однако не всякую программу можно специфицировать таким образом. Имеется класс программ реального времени, в частности реактивных систем, где программа представлена в виде набора параллельных взаимодействующих процессов. Примерами таких программ являются протоколы и телекоммуникационные системы. В целях спецификации и реализации данного класса программ язык P расширен средствами для описания процессов, передачи сообщений и порождения процессов, работающих параллельно с процессом-родителем.

ОПРЕДЕЛЕНИЕ-ПРОЦЕССА ::=
process ИМЯ-ПРОЦЕССА [**(** [ОПИСАНИЯ-АРГУМЕНТОВ] **:**
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ **)**]
{ ТЕЛО-ПРОЦЕССА [МЕТКА **:**]**}**
[**spec** ВНЕШНЯЯ-СПЕЦИФИКАЦИЯ-ПРОЦЕССА]
ИМЯ-ПРОЦЕССА ::= ИДЕНТИФИКАТОР

Процесс может быть бесконечным, и в этом случае он не имеет результатов. Результаты конечного завершающегося процесса определяются так же, как у гиперфункции. Конструкция вида имя: в теле процесса является описанием локальной метки, на которую возможен переход из операторов процесса. Определяемый процесс имеет структуру машины конечных состояний в виде гиперграфа, в котором гипердугами являются подпроцессы или гиперфункции. Использование аппарата гиперфункций обеспечивает большую гибкость в декомпозиции процессов.

ВНЕШНЯЯ-СПЕЦИФИКАЦИЯ-ПРОЦЕССА описывает эффект функционирования процесса на внешнее окружение, в рамках которого существует процесс. В общем случае этот эффект невозможно выразить с помощью предусловия и постусловия. В качестве языков внешней спецификации процессов ранее использовались язык алгебры процессов CCS [8] Р. Милнера и язык спецификации TLA+ [9] Л. Лампорта. Язык внешней спецификации в данном документе не фиксируется. Наиболее адекватным и проработанным представляется язык SAL [10].

ТЕЛО-ПРОЦЕССА ::= ОПИСАНИЕ-ПЕРЕМЕННЫХ [**;** ТЕЛО-ПРОЦЕССА] |
ВОЗМОЖНО-ПОМЕЧЕННЫЙ-ОПЕРАТОР [**;** ТЕЛО-ПРОЦЕССА]
ВОЗМОЖНО-ПОМЕЧЕННЫЙ-ОПЕРАТОР ::= [МЕТКА **:**] ОПЕРАТОР

Переменные и метки, описанные в теле процесса, считаются локальными в данном процессе. Описания и операторы блока исполняются в порядке их следования. Оператор перехода изменяет последовательность выполнения. Допускаются циклы в теле процесса.

Языковые конструкции ОПЕРАТОР, ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ и ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА, определенные выше, имеют доопределение для процессов.

ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА ::= ... | [ОБЪЕКТ-КЛАССА .] ИМЯ-ПРЕДИКАТА
ОБЪЕКТ-КЛАССА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ

Конструкция ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА используется в вызове предиката для указания вызываемого предиката.

ОПЕРАТОР-РАБОТЫ-С-ПРОЦЕССАМИ ::=
ВЫЗОВ-ПРОЦЕССА | ЗАПУСК-НЕЗАВИСИМОГО-ПРОЦЕССА |
ПАРАЛЛЕЛЬНАЯ-КОМПОЗИЦИЯ-ПРОЦЕССОВ | ПРИЕМ-СООБЩЕНИЯ |
ПОСЫЛКА-СООБЩЕНИЯ | ЗАЩИЩЕННЫЙ-ОПЕРАТОР | ОПЕРАТОР-ПЕРЕХОДА
ВЫЗОВ-ПРОЦЕССА ::=

[ОБЪЕКТ-КЛАССА **.**] ИМЯ-ПРОЦЕССА **(** [АРГУМЕНТЫ] (**:** РЕЗУЛЬТАТЫ-ВЕТВИ)*] **)**

ОБЪЕКТ-КЛАССА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ

Процесс, запускаемый на исполнение ВЫЗОВОМ-ПРОЦЕССА, должен иметь соответствующее определение процесса. Если определение процесса принадлежит некоторому классу, то вызов процесса реализуется через ОБЪЕКТ-КЛАССА.

ПАРАЛЛЕЛЬНАЯ-КОМПОЗИЦИЯ-ПРОЦЕССОВ ::= ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР

По синтаксису параллельная композиция процессов аналогична параллельному оператору. Параллельная композиция задает параллельное исполнение процессов, вызовы которых указаны в композиции. Взаимодействие между процессами реализуется посредством приема и передачи сообщений.

ОПИСАНИЕ-СООБЩЕНИЯ ::= **message** ИМЯ-СООБЩЕНИЯ [**(** СПИСОК-ТИПОВ **)**] |
queue ИМЯ-СООБЩЕНИЯ [**(** СПИСОК-ТИПОВ **)**]

ИМЯ-СООБЩЕНИЯ ::= ИДЕНТИФИКАТОР

СПИСОК-ТИПОВ ::= ИЗОБРАЖЕНИЕ-ТИПА [**,** СПИСОК-ТИПОВ]

Взаимодействие процесса с окружением реализуется посредством сообщений. Сообщение содержит (возможно, пустой) набор значений, типы которых указываются в параметрах описания сообщения.

ПОСЫЛКА-СООБЩЕНИЯ ::=
ИМЯ-ПРОЦЕССА> **!** ИМЯ-СООБЩЕНИЯ [**(** СПИСОК-ВЫРАЖЕНИЙ **)**] |
ИМЯ-СООБЩЕНИЯ **!** [**(** СПИСОК-ВЫРАЖЕНИЙ **)**]

Оператор посылает сообщение с набором значений, вычисленных СПИСКОМ-ВЫРАЖЕНИЙ. Если сообщение определено описанием **message**, то выполнение оператора не будет завершено до тех пор, пока это сообщение не будет принято получателем. Для сообщения, определенного описанием **queue**, оператор вставляет сообщение в конец очереди посланных сообщений и завершает свою работу.

ПРИЕМ-СООБЩЕНИЯ ::= **receive** СООБЩЕНИЕ |
ПРИЕМ-СООБЩЕНИЯ-ОБЩЕГО-ВИДА

СООБЩЕНИЕ ::= ИМЯ-СООБЩЕНИЯ [**(** СПИСОК-ПЕРЕМЕННЫХ **)**]

Допустим, что процесс, исполняющий оператор receive СООБЩЕНИЕ, получил сообщение с именем, указанным в операторе. Исполнение оператора заключается в присваивании перечисленным переменным соответствующих значений, поставляемых сообщением. Если исполняемый процесс не получил сообщения с указанным именем, то исполнение оператора приостанавливается (блокируется) до поступления требуемого сообщения.

ПРИЕМ-СООБЩЕНИЯ-ОБЩЕГО-ВИДА ::= **receive {** СООБЩЕНИЕ **:** ОПЕРАТОР
(**or** СООБЩЕНИЕ **:** ОПЕРАТОР)*
[**after** ВРЕМЯ-ОЖИДАНИЯ **:** ОПЕРАТОР]
}

ВРЕМЯ-ОЖИДАНИЯ ::= ВЫРАЖЕНИЕ

В результате выполнения оператора общего вида реализуется прием одного из указанных сообщений: того, которое приходит первым. При этом соответствующим переменным присваиваются значения параметров сообщения. Далее исполняется ОПЕРАТОР, следующий за «:» после пришедшего сообщения. Оператор общего вида также является блокирующим: исполнение текущего процесса приостанавливается до поступления любого из указанных сообщений. Если в операторе присутствует подконструкция **after**, то ожидание поступления одного из перечисленных сообщений ограничено по времени. Время ожидания имеет тип **nat** и определяет значение времени в миллисекундах, по истечении которого срабатывает ОПЕРАТОР, указанный завершителем **after**.

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ ::= ... | СООБЩЕНИЕ

Конструкция СООБЩЕНИЕ в любой позиции логического выражения трактуется как неблокирующий прием сообщений. Исполнение конструкции осуществляется следующим образом. В очереди пришедших сообщений исполняемого процесса ищется сообщение с

именем, указанным в конструкции СООБЩЕНИЕ. Если сообщение обнаружено, оно вычеркивается из очереди. При этом значения параметров присваиваются соответствующим переменным, перечисленным в круглых скобках. Результатом исполнения конструкции СООБЩЕНИЕ является значение **true**. Если требуемое сообщение не обнаружено в очереди, исполнение конструкции СООБЩЕНИЕ завершается значением **false**.

Динамическое создание нового процесса и его запуск реализуется через объект класса. Объект (или значение) типа "класс" является структурой, составленной из полей класса. Описание класса определяется следующим образом:

ОПРЕДЕЛЕНИЕ-КЛАССА ::=
class ИМЯ-КЛАССА [**extends** ИМЯ-СУПЕРКЛАССА] **{** ТЕЛО-КЛАССА **}**
ИМЯ-КЛАССА ::= ИДЕНТИФИКАТОР
ТЕЛО-КЛАССА ::= ОПИСАНИЕ-КЛАССА [**;** ТЕЛО-КЛАССА]
ОПИСАНИЕ-КЛАССА ::=
ОПИСАНИЕ-ТИПА | ОПИСАНИЕ-ПЕРЕМЕННЫХ | ОПИСАНИЕ-СООБЩЕНИЯ |
ОПРЕДЕЛЕНИЕ-ПРОЦЕССА | ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА |
ОПИСАНИЕ-КОНСТРУКТОРА-КЛАССА
ОПИСАНИЕ-КОНСТРУКТОРА-КЛАССА ::=
ИМЯ-КЛАССА **(**[<ОПИСАНИЯ-АРГУМЕНТОВ>]**) {** ТЕЛО-КОНСТРУКТОРА **}**
ТЕЛО-КОНСТРУКТОРА ::= [ТЕЛО-БЛОКА]

Доступ к элементу класса вне описания класса реализуется через объект класса, т. е. конструкцией объект.имя, где имя – имя элемента класса. Доступ к элементу класса в теле класса осуществляется непосредственно по имени элемента. Для обозначения объекта, определяемого данным классом, используется имя **this**.

Отметим, что классы не интегрированы с системой типов языка P.

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ ::= ... | СОЗДАНИЕ-ОБЪЕКТА
СОЗДАНИЕ-ОБЪЕКТА ::= **new** ИМЯ-КЛАССА **(** [<АРГУМЕНТЫ>] **)**

Операция СОЗДАНИЯ-ОБЪЕКТА реализует вызов конструктора, параметры которого соответствуют списку аргументов. Результатом операции является новый объект (значение) типа ИМЯ-КЛАССА. При исполнении конструктора создается объект – структура, полями которой являются переменные, описанные в теле класса. Для созданного объекта исполняются ОПИСАНИЯ-ПЕРЕМЕННЫХ, находящиеся в теле класса и содержащие инициализацию переменных. Далее исполняется тело конструктора.

Если для операции **new** ИМЯ-КЛАССА( ) нет соответствующего конструктора с пустым списком аргументов, то такой конструктор считается определенным и его тело пусто.

ИМЯ-СУПЕРКЛАССА ::= ИМЯ-КЛАССА

При наличии подконструкции **extends** ИМЯ-СУПЕРКЛАССА определяемый класс является расширением суперкласса, представленного описателем **extends**. Структура, создаваемая для определяемого класса, содержит поля суперкласса, за которыми следуют поля определяемого класса. Конструктор класса содержит явный или неявный вызов соответствующего конструктора для суперкласса.

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ ::= ... | ДИНАМИЧЕСКОЕ-СОЗДАНИЕ-ПРОЦЕССА

ДИНАМИЧЕСКОЕ-СОЗДАНИЕ-ПРОЦЕССА ::=
new ИМЯ-ПРОЦЕССА **(** [<АРГУМЕНТЫ>] **)**

Данная операция реализует создание нового независимого процесса и его запуск на исполнение. Процесс с указанным именем должен быть элементом некоторого класса. В начале тела процесса находится явный или неявный вызов конструктора класса. При исполнении операции **new** срабатывает вызов конструктора в начале тела процесса. Результатом вызова конструктора является новый объект класса. Далее создается новый процесс и запускается на исполнение параллельно с текущим исполняемым процессом. Программой созданного процесса является тело процесса, а глобальными данными – поля созданного объекта. На этом

исполнение операции **new** завершается. Ее результатом является созданный объект. Текущий процесс продолжает исполняться параллельно с созданным процессом. Объект, являющийся результатом операции **new**, используется в качестве указателя созданного процесса в операторах обмена сообщениями.

ЗАЩИЩЕННЫЙ-ОПЕРАТОР ::=
with СПИСОК-РАЗДЕЛЯЕМЫХ-ПЕРЕМЕННЫХ **{** ОПЕРАТОР **}**
СПИСОК-РАЗДЕЛЯЕМЫХ-ПЕРЕМЕННЫХ ::= СПИСОК-ПЕРЕМЕННЫХ

Взаимодействие между параллельными процессами возможно через разделяемые переменные, доступные для нескольких процессов и являющиеся глобальными по отношению к ним. Доступ к разделяемым переменным разрешен только внутри защищенного оператора. При исполнении ОПЕРАТОРА доступ к перечисленным разделяемым переменным блокируется. Блокировка снимается при завершении исполнения ОПЕРАТОРА. Если при попытке исполнить ЗАЩИЩЕННЫЙ-ОПЕРАТОР одна из разделяемых переменных оказалась блокирована другим процессом, исполнение текущего процесса, исполняющего ЗАЩИЩЕННЫЙ-ОПЕРАТОР, приостанавливается до момента снятия блокировки с разделяемой переменной.

6.11. Императивное расширение

Императивное расширение языка P определяет дополнительные языковые конструкции, возникающие в программе в результате проведения трансформаций предикатной программы (см. введение). Использование этих конструкций в исходной программе недопустимо.

ЗАГОЛОВОК-ПРЕДИКАТА ::=
ИМЯ-ПРЕДИКАТА **(** [ОПИСАНИЯ-АРГУМЕНТОВ] [**:** ОПИСАНИЯ-РЕЗУЛЬТАТОВ] **)**
ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ ::=
ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА **(** [АРГУМЕНТЫ] [**:** РЕЗУЛЬТАТЫ] **)**

В императивном расширении для предиката-функции допускается отсутствие результатов. Это отражено в приведенных определениях, расширяющие определения, данные в разд. 6.3.1. Отсутствие результатов возможно, например, после склеивания результатов с глобальными переменными.

Определим дополнительные операторы императивного расширения.

ОПЕРАТОР-ИМПЕРАТИВНОГО-РАСШИРЕНИЯ ::=
break | ОПЕРАТОР-FOR | ОПЕРАТОР-ЕСЛИ

В императивной программе, полученной в результате трансформации, в позиции оператора исходной предикатной программы может также находиться оператор императивного расширения. Отметим, что групповой оператор присваивания, возникающий при подстановке тела предиката на место вызова предиката, является частным случаем оператора присваивания, когда в левой части – мультипеременная, а в правой – мультивыражение.

ОПЕРАТОР-FOR ::= **for (** ЗАГОЛОВОК-ЦИКЛА **) {** ОПЕРАТОР [**;** ОПЕРАТОР] **}**
ЗАГОЛОВОК-ЦИКЛА ::= [[ИЗОБРАЖЕНИЕ-ТИПА] ПАРАМЕТР-ЦИКЛА **=** ВЫРАЖЕНИЕ] **;**
[УСЛОВИЕ-ЗАВЕРШЕНИЯ] **;**
[ПЕРЕСЧЕТ-ПАРАМЕТРА]
ПАРАМЕТР-ЦИКЛА ::= ИДЕНТИФИКАТОР
УСЛОВИЕ-ЗАВЕРШЕНИЯ ::= ВЫРАЖЕНИЕ
ПЕРЕСЧЕТ-ПАРАМЕТРА ::= ОПЕРАТОР

В императивном расширении используются также операторы перехода и помеченные операторы. Синтаксис и семантика определены в разд. 10.

Семантика цикла **for** соответствует языку C++. Выход из цикла **for** может также быть реализован оператором **break** из тела цикла.

ЦИКЛ-WHILE ::= **while (** ВЫРАЖЕНИЕ **)** ОПЕРАТОР

Тело цикла **while** исполняется, пока логическое выражение в его заголовке истинно.

```
sum (list (int) a : int r) {
  if (len (a) > 0) {
    r = a [0];
    int i = 1;
    while (i < len (a)) {
      r = r + a [i];
      i = i + 1;
    }
  } else {
    r = 0;
  }
}
```

Пример 18. Использование цикла **while**

Данная программа может получиться в результате применения первых трех трансформаций: склеивания переменных, замены хвостовой рекурсии циклом и подстановки определения предиката на место вызвова.

Циклы **for** и **while** часто используектся для представления циклов, реализуемых с помощью операторов перехода, в целях улучшения структуры программы.

Список литературы

1. *Шелехов В. И.* Введение в предикатное программирование. Новосибирск, 2002. 82 с. (Препр. / ИСИ СО РАН № 100).

2. *Шелехов В. И.* Язык предикатного программирования P. Новосибирск, 2002. 40 с. (Препр. / ИСИ СО РАН № 101).

3. *Шелехов В. И.* Предикатное программирование: основы, язык, технология // Методы предикатного программирования. Новосибирск, ИСИ СО РАН. 2003. С. 7–15.

4. *Шелехов В. И.* Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52 с. (Препр. / ИСИ СО РАН № 115).

5. *Шелехов В. И.* Язык спецификации процессов // Методы предикатного программирования. Новосибирск, ИСИ СО РАН. 2006. Вып. 2. С. 17–34.

6. *Шелехов В. И.* Модель корректности программ на языке исчисления вычислимых предикатов. Новосибирск, 2007. 51 с. (Препр. / ИСИ СО РАН № 145).

7. PVS Specification and Verification System. URL: http://pvs.csl.sri.com/

8. *Milner R.* A Calculus of Communicating Systems // LNCS. Springer Verlag, 1980. Vol. 94.

9. *Lamport L.* Specifying concurrent systems with TLA+. Calculational System Design. Series F: Computer and Systems Sciences. Amsterdam: IOS Press. 1999. N. 173. P. 183–247.

10. *Moura L., Owre S., and Shankar N.* The SAL Language Manual. SRI International. URL: http://sal.csl.sri.com/doc/language-report.pdf.

7. Технология предикатного программирования

Типичный способ реализации языка программирования заключается в реализации транслятора с языка программирования на язык команд целевой ЭВМ или на некоторый другой реализованный язык программирования, а также реализации поддержки исполнения. В классификации, представленной в конце разд. 1.1, это соответствует реализации виртуального процессора. Применение оптимизации в процессе трансляции программ для императивных языков позволяет получать вполне эффективные программы. Однако для функциональных языков не удается достичь приемлемой эффективности даже с помощью изощренной оптимизации. Подобная перспектива неэффективности оттранслированной программы ожидает также и реализацию языка предикатного программирования P методом прямой автоматической трансляции.

Для языка P разрабатывается метод управляемой полуавтоматической трансляции применением последовательности трансформаций, преобразующих предикатную программу в эффективную императивную программу на императивном расширении языка P (см. разд. 6.11). Базовыми *трансформациями* являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

Для всех определений предикатов сначала применяется склеивание переменных, затем замена хвостовой рекурсии циклом. Тела предикатов с устраненной рекурсией подставляются на место вызовов. После этого можно провести другие упрощающие преобразования, в частности константные вычисления. Наконец, структурные объекты (списки, деревья и др.) кодируются посредством массивов и указателей. С помощью данных трансформаций можно получить программу предельной эффективности, однако трудно это сделать автоматически. На втором этапе полученная императивная программа конвертируется на любой из императивных языков: C, C++, ФОРТРАН и др. В настоящее время при программировании на языке P применяется метод ручной трансформации предикатной программы.

Трансформация замены хвостовой рекурсии циклом является весьма эффективной, поскольку открывает возможность применения серии других оптимизирующих преобразований. Однако рекурсия в наиболее простом решении задачи обычно не является хвостовой. Существует ряд подходов по приведению рекурсии к хвостовому виду, как правило, автоматическими преобразованиями. В технологии предикатного программирования предлагается универсальный метод обобщения исходной задачи для получения решения с хвостовой формой рекурсии. Этот метод нередко оказывается ключевым для эффективной реализации предикатных программ.

В программировании регулярно возникают ситуации, когда традиционными конструкциями языка программирования невозможно адекватно представить некоторый фрагмент алгоритма. Предлагаемый аппарат гиперфункций позволяет гибко декомпозировать программу произвольным образом.

7.1. Подстановка определения предиката на место вызова

Пусть A(x: y) { S } – определение предиката на императивном расширении языка P, а A(e: z) – вызов предиката в теле некоторого другого предиката B. Здесь x, y, z обозначают списки переменных, а e – список выражений. *Подстановка определения предиката на место вызова* A(e: z) есть замена вызова следующей композицией:

$$| x | = | e |; \{ S \}; | z | = | y | \ . \qquad (7.1)$$

Конструкции | x |, | z | и | y | являются мультипеременными, а | e | – мультивыражением (см. разд. 6.6). В результате подстановки переменные наборов x и y становятся локальными переменными определения предиката B, в котором находился вызов A(e: z). В результате подстановки не должно возникать коллизий с именами переменных предиката B. В общем случае проведению подстановки предшествует предварительное систематическое переименование переменных. Нетрудно показать, что исполнение последовательности (7.1) эквивалентно исполнению вызова A(e: z) в соответствии с операционной семантикой (4.14) и семантикой выражений как аргументов вызова, определенной в разд. 5.4.

Описанная трансформация подстановки определения реализуется другим способом, нежели похожая на нее операция подстановки определения на место вызова, введенная в разд. 5.1 для программы на языке ССР. Применение операции подстановки, введенной в разд. 5.1, ухудшит программу в случае замены всех вхождений аргумента в теле предиката на выражение, являющееся соответствующим параметром вызова.

Оператор присваивания вида | x | = | e | называется *групповым оператором присваивания*. В соответствии с операционной семантикой (см. разд. 6.6) вычисление выражений, входящих в набор e, проводится параллельно. Эффективная реализация такого параллелизма возможна на процессорах с архитектурой широких команд. На других архитектурах параллельная реализация неэффективна, и поэтому замена группового оператора набором обычных операторов присваивания становится неизбежной. Это преобразование называется *раскрытием* группового оператора присваивания. В общем случае раскрытие группового оператора возможно при использовании дополнительных промежуточных переменных. Например, раскрытие оператора |a ,b| = |b, a| реализуется операторами t = a; a = b; b = t с использованием дополнительной переменной t. В большинстве случаев раскрытие возможно без использования дополнительных переменных; иногда достаточно поменять местами операторы присваивания. Например, раскрытие |a, b| = |c, a| реализуется присваиваниями: b = a; a = c.

Пусть набор x состоит из переменных $x_1, x_2, \ldots, x_n$; а e определяет набор выражений $e_1, e_2, \ldots, e_n$. Допустим, e_j является переменной для некоторого j. В соответствии с соглашением об отсутствии коллизий переменная e_j должна быть отлична от x_j. В подавляющем большинстве случаев замена x_j на e_j дает эквивалентную программу. Данная замена (склеивание переменной x_j с e_j) уменьшает число переменных на единицу и позволяет удалить копирование $x_j = e_j$ в составе группового оператора | x | = | e |. Склеивание переменных x_j и e_j корректно за исключением случая, когда x_j перевычисляется внутри S (что возможно как результат склеивания x_j с другими переменными (см. разд. 7.3)), а переменная e_j используется не только в вызове A(e: z), но и после него. Имеется еще одно условие: до проведения замены x_j на e_j переменная e_j не должна встречаться в измененной версии S в (7.1); последнее возможно в случае, когда проведено склеивание переменных x_k и e_k, причем e_k совпадает с e_j.

Аналогичным образом проводится склеивание результирующих переменных набора y с переменными набора z. Поскольку переменные, входящие в набор z, должны быть различными, то склеивание результирующих переменных всегда возможно. При склеивании результатов устраняется групповой оператор | z | = | y |. Отмеченные склеивания (замену x_j на e_j и y_j на z_j) можно не проводить, если склеиваемые переменные обозначены одинаковыми идентификаторами в исходной программе. Поэтому естественной является следующая стратегия именования в процессе программирования на языке P. Если имеется вызов A(e: z) и требуется запрограммировать определение предиката A, то в качестве результирующих параметров y в определении A выбираются переменные набора z. Переменные в составе набора e становятся соответствующими аргументами определения A с учетом ограничений, указанных выше.

7.2. Замена хвостовой рекурсии циклом

Подстановка определения нерекурсивного предиката на место вызова является эффективной при наличии в программе одного вызова. Подстановка определения рекурсивного предиката усложняет программу и поэтому сомнительна, хотя иногда может принести пользу; в любом случае такое преобразование считается экзотическим. Для рекурсивно определяемого предиката эффективным является другое преобразование – замена хвостовой рекурсии циклом. Тем не менее, оказывается, это преобразование является специальным случаем подстановки определения предиката на место вызова.

Существует специальный случай рекурсии, называемый *хвостовой рекурсией* (tail-рекурсией в языке Лисп [1]), когда можно заменить рекурсию циклом. Рекурсивный вызов предиката определяет хвостовую рекурсию, если:

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

Через last(S) обозначим множество последних исполняемых конструкций в операторе S. Тогда: last(A; B) = last(B), last(**if** (C) A **else** B) = last(A) ∪ last(B), last(D(e: z)) = {D(e: z)}, last(A || B) = ∅. Однако если параллельный оператор A || B реализуется последовательным исполнением A и B, например, как B; A, то last(A || B) = last(A).

Понятие хвостовой рекурсии проиллюстрируем на примерах 5.1 и 5.2 (см. разд. 5.5). В программе умножения через сложение (см. пример 5.1)

Умн(**nat** a, b**:** **nat** c) { **if** (a = 0) c = 0 **else** c = b + Умн(a – 1, b) } (7.2)

рекурсивный вызов Умн(a – 1, b) не является хвостовым, поскольку после завершения исполнения вызова исполняется операция “+”. Хвостовой является рекурсия для каждого из двух рекурсивных вызовов в программе вычисления наибольшего общего делителя (см. пример 5.2):

D(**nat** a, b**:** **nat** c) { **if** (a = b) c = a **else if** (a < b) D(a, b – a**:** c) **else** D(a – b, b**:** c) }

Допустим, имеется рекурсивное определение предиката A(x: y) { S } и вызов A(e: z) с хвостовой рекурсией внутри оператора S. Тогда этот вызов должен иметь вид A(e: y), т. е. z = y, поскольку хвостовой вызов с другим набором результатов, кроме y, смысла не имеет. Подставим определение предиката A на место вызова A(e: y). В соответствии с изложенным в конце разд. 7.1 результатом подстановки определения предиката A является | x | = | e |; { S }. Обозначим через S′ оператор, полученный заменой в S подстановкой определения на место вызова A(e: y). Очевидно, можно заменить в S′ второе вхождение S передачей управления на начало оператора S′. В итоге определение предиката A преобразуется к виду: A(x: y) {M: S″ }, где S″ получается из S заменой вызова A(e: y) парой операторов: | x | = | e |; **goto** M. Данное преобразование есть трансформация *замены хвостовой рекурсии циклом*.

Если в рекурсивном определении предиката A все рекурсивные вызовы имеют вид хвостовой рекурсии, то применение трансформации ко всем этим вызовам преобразует тело предиката в цикл, а определении предиката A – в нерекурсивное определение. После этой трансформации становится эффективной постановка определения предиката A на место вызова A в теле другого предиката. А после проведения подстановки вызова A становится возможной серия других преобразований.

Применение трансформации замены хвостовой рекурсии циклом покажем для программы вычисления наибольшего общего делителя. Трансформация двух рекурсивных вызовов предиката D дает следующую программу:

```
D(nat a, nat b: nat c) {
      M:    if (a = b) c = a
            else  if (a < b) |a, b| = |a, b – a|; goto M
                  else |a, b| = |a – b, b|; goto M
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **for**. Получим:

```
D(nat a, nat b: nat c) {
      for ( ; ; ) {
            if (a = b) {c = a; break; }
            if (a < b) b = b – a
            else a = a – b
      }
}
```

Замена фрагмента с операторами перехода на цикл **for** не меняет процесса исполнения. Преобразование такого рода, улучшающее структуру программы, будем называть *оформлением.*

7.3. Склеивание переменных

Трансформация *склеивания переменных* есть замена в определении предиката нескольких переменных одной. Трансформация определяет список имен переменных. Переменные из указанного списка переменных заменяются первой переменной. Например, склеивание переменных a, b и c представляет систематическую замену всех вхождений имен b и c на имя a. Необходимо соблюдение условий корректности, гарантирующих эквивалентность определений предиката до и после проведения склеивания. Склеивание переменных является первой трансформацией среди перечисленных в начале разд. 7 четырех видов трансформаций. В результате склеивания уменьшается число переменных, программа становится короче и быстрее. Значительный эффект достигается при склеивании структурных переменных, таких как массивы и списки, поскольку склеивание обычно позволяет избежать копирования структур.

Задача склеивания переменных в данной постановке вряд ли может стать актуальной для оптимизации функциональных программ: в определении функции нет результирующих переменных, и там можно было бы рассматривать лишь склеивание локальных и исходных переменных. Эта задача не возникает для императивных программ, поскольку практически все рассматриваемые здесь склеивания обычно реализуются программистом в императивной программе.

Склеивание переменных рассматривалось ранее в рамках задачи экономии памяти в классических работах А. П. Ершова, С. С. Лаврова, В. В. Мартынюка. Склеивание переменных определялось как выбор переобозначения аргументов и результатов из заданного множества корректных переобозначений, которое позволяет в наибольшей степени уменьшить объем необходимой памяти [2]. В оптимизирующей трансляции императивных программ склеивание переменных обычно реализуется на основе анализа локальных свойств программы [3; 4]. Отметим, что необходимость склеивания переменных обусловлена также тем, что в применяемых методах трансляции конструкций (особенно выражений) происходит включение в программу большого числа дополнительных промежуточных переменных.

В отличие от задачи экономии памяти [2] склеиванию подлежат только те переменные, между которыми имеется информационная связь в определении предиката. Кроме того, склеиваются лишь переменные одного типа, однако допускается различие в параметрах типа. Алгоритм автоматического оптимального склеивания переменных описан в работе [5]. В каждом операторе программы определяются аргументы и результаты оператора. Результат оператора может быть склеен с любым аргументом соответствующего типа при условии, что

аргумент не используется далее после завершения исполнения оператора. В работе [5] не рассматриваются сложные формы склеивания переменных структурных типов, когда несколько структурных переменных размещаются внутри некоторой другой переменной. Пример сложной формы склеивания иллюстрируется в разд. 7.6.

7.4. Метод обобщения исходной задачи

Трансформация замены хвостовой рекурсии циклом является весьма эффективной, поскольку устранение рекурсии позволяет провести подстановку определения на место вызова, после чего открывается возможность применения серии других оптимизирующих преобразований. Однако рекурсия в наиболее простом решении задачи обычно не является хвостовой. Существует ряд подходов по приведению рекурсии к хвостовому виду, как правило, автоматическими преобразованиями. Однако автоматическое приведение определения предиката к виду хвостовой рекурсии может быть оправдано лишь в том случае, когда все дальнейшие преобразования совершаются в автоматическом режиме. По нашей точке зрения, неправильно считать приведение к виду хвостовой рекурсии оптимизирующим преобразованием, реализуемым в процессе трансляции. Данное преобразование меняет алгоритм решения задачи, и поэтому должно применяться на стадии выбора алгоритма и программирования.

В качестве адекватной техники программирования предлагается универсальный метод *обобщения исходной задачи* для получения решения с хвостовой формой рекурсии. Этот метод часто оказывается ключевым для эффективной реализации предикатных программ.

Дадим иллюстрацию метода на примере 5.1 программы умножения через сложение. Как отмечено выше, рекурсия в программе (7.2) не является хвостовой, поскольку после рекурсивного вызова Умн(a – 1, b) в операторе c = b + Умн(a – 1, b) реализуется сложение с b. Чтобы сделать рекурсию хвостовой, необходимо внести сложение с b в рекурсивный вызов. Типичный прием – использование дополнительного параметра d в качестве накопителя при суммировании. Рассмотрим обобщенную задачу реализации умножения $a * b$ через сложение с использованием накопителя d. Представим задачу следующим предикатом:

УмнО(**nat** a, b, d**:** **nat** c) **pre** $a \geq 0$ & $b \geq 0$ & $d \geq 0$ **post** $c = a * b + d$;

Тогда Умн(a, b**:** c) ≡ УмнО(a, b, 0**:** c), и поэтому корректным является следующее определение предиката:

```
Умн(nat a, b:  nat c)                                          (7.3)
 pre a ≥ 0 & b ≥ 0
 { УмнО(a, b, 0: c) }
 post c = a * b;
```

Формальное доказательство корректности (при очевидной однозначности и реализуемости) проводится доказательством истинности следующих правил, являющихся упрощенными аналогами правил LS8 и LS9.

Правило LSУ8. $P_{Умн}(a, b) \vdash$ **true** (в качестве предусловия константы 0).

Правило LSУ9. $P_{Умн}(a, b)$ & $Q_{Умн}(a, b, c) \vdash P_{УмнО}(a, b, 0)$ & $Q_{УмнО}(a, b, 0, c)$. □

Предикат УмнО имеет следующее определение:

```
УмнО(nat a, b, d:  nat c)                                      (7.4)
 pre a ≥ 0 & b ≥ 0 & d ≥ 0
 { if (a = 0) c = d else УмнО(a – 1, b, d + b: c) }
 post c = a * b + d;
```

Доказательство корректности определения проводится так же, как и для определения Умн в примере 5.1 (см. разд. 5.5). □

Проиллюстрируем технику трансформации программы умножения, представленную определениями предикатов Умн и УмнО. На первом этапе применяется трансформация склеивания переменных. Аргумент d склеивается с результатом c, поскольку c информаци-

онно связан с d в обеих альтернативах условного оператора. Отметим, что дополнительная переменная, введенная при обобщении задачи, обычно подлежит склеиванию с некоторым результатом. Склеивание d с c будем обозначать $c \leftarrow d$. В результате склеивания получим следующее определение:

```
УмнО(nat a, b, c:  nat c) { if (a = 0) c = c else УмнО(a – 1, b, c + b: c) } .
```

Далее применяется трансформация замены хвостовой рекурсии циклом

```
УмнО(nat a, b, c:  nat c) {M: if (a = 0) else |a, b, c| = |a – 1, b, c + b|; goto M } .
```

После раскрытия группового оператора и оформления цикла получим

```
УмнО(nat a, b, c:  nat c) {for (; a <> 0 ;) {a = a – 1; c = c + b} } .        (7.5)
```

Последней является трансформация подстановки определения (7.5) на место вызова в (7.3). В соответствии с определением подстановки (7.1) получим:

```
|a, b, c| = | a, b, 0 |; for (; a <> 0 ;) {a = a – 1; c = c + b} .
```

Раскрывая групповой оператор, получим итоговую программу:

```
c = 0; for (; a <> 0 ;) {a = a – 1; c = c + b} .
```

Результатом проведенной серии трансформаций предикатной программы (7.3), (7.4) является представленная императивная программа, по качеству не хуже запрограммированной вручную. Разумеется, данная программа далеко не самая эффективная. В процессорах ЭВМ применяются более изощренные алгоритмы, использующие особенности двоичного представления чисел a и b. В предикатном программировании эффективность конечной программы в значительной степени зависит от выбора эффективного алгоритма, превращаемого в эффективную программу применением последовательности трансформаций.

7.5. Трансформация кодирования структурных объектов

Трансформация *кодирования структурных объектов* реализует представление используемых в программе структурных типов (списков, деревьев и др.) посредством структур более низкого уровня, таких как массивы и указатели. Операции с используемыми в программе структурными типами выражаются в терминах структур низкого уровня. Трансформация реализует систематическую замену вхождений таких операций в программе на соответствующие эквивалентные конструкции, представленные на языке структур низкого уровня. Применение данной трансформации иллюстрируется на следующем примере суммирования списка.

Пример 7.1. Программа суммирования sum(s: c) элементов списка s.

Отметим, что более типичной является программа суммирования элементов массива. Однако массив является объектом логики второго порядка. Список более предпочтителен по сравнению с массивом, поскольку операции со списком выражаются в рамках логики первого порядка.

Спецификацию программы суммирования определим рекурсивным предикатом:

```
SUM(s, c) ≅ s = nil ? c = 0 : ∃ real a. SUM(s.cdr, a) & c = s.car + a .        (7.6)
```

Простейшая программа суммирования тождественна спецификации (7.6).

```
type seqR = list (real);
sum(seqR s: real c)                                                          (7.7)
      { if (s = nil ) c = 0 else c = s.car + sum(s.cdr) }
      post SUM(s, c);
```

Доказательство корректности определения sum реализуется на основе правил LR3 и LR4 по схеме, сформулированной в конце разд. 5.5. Очевидна реализуемость спецификации и однозначность правой части определения. Переменная s является индуктивной. На множестве списков определим отношение порядка: $u \sqsubseteq s \cong \exists v.\ v + u = s$. Строгий частичный порядок $\sqsubset$ определяется на основе $\sqsubseteq$. Тогда истинно $s.cdr \sqsubset s$. Минимальный элемент nil покрывается условием s = nil в условном операторе. Операции s.car и s.cdr имеют предусловие $s \neq nil$. Это предусловие истинно для альтернативы **else**, и поэтому согласно лемме 5.8 достаточно

доказать истинность оператора c = s.car + sum(s.cdr). В соответствии с индуктивным предположением мы можем заменить sum на SUM. □

Рекурсивная программа (7.7) неэффективна, поскольку рекурсия не является хвостовой. Введем накопитель d. Рассмотрим обобщенную задачу суммирования:

sumG(seqR s, **real** d**: real** c) **post** ∃e. SUM(s, e) & c = e + d;

Ввиду истинности тождества sum(s**:** c) ≡ sumG(s, 0**:** c) корректно следующее определение предиката:

```
sum(seqR s: real c)                                              (7.8)
        { sumG(s, 0: c) }
        post ∃ real e. SUM(s, e) & c = e + d;
```

Предикат sumG имеет следующее определение:

```
sumG(seqR s, real d: real c)                                     (7.9)
        { if (s = nil ) c = d else sumG(s.cdr, d + s.car : c) }
        post ∃ real e. SUM(s, e) & c = e + d;
```

Доказательство корректности определения sumG реализуется аналогично доказательству корректности определения (7.7). Поскольку s.cdr ⊏ s, то вызов sumG в теле (7.9) может быть заменен постусловием sumG. Истинность полученной формулы доказывается применением определения (7.6). □

Хвостовая рекурсия в определении sumG дает возможность провести для программы (7.8), (7.9) следующую серию трансформаций, аналогичную примененной для задачи Умн. Проведем склеивание c ← d.

```
sumG(seqR s, real c: real c) { if (s = nil ) c = c else sumG(s.cdr, c + s.car : c) }
```

Заменим хвостовую рекурсию циклом. Отметим, что при раскрытии группового оператора необходимо будет поменять порядок присваивания параметрам s и c.

```
sumG(seqR s, real c: real c) { for (; s <> nil; ) { c = c + s.car; s = s.cdr }
```

Подставим полученное определение в (7.8).

```
c = 0; for (; s <> nil; ) { c = c + s.car; s = s.cdr }           (7.10)
```

Программа (7.10) использует три операции со списками. Непосредственная реализация оператора s = s.cdr приводит к копированию большей части списка. Поэтому такая программа неэффективна.

Для программы (7.10) применяется трансформация кодирования списка вырезкой массива. При этом разные значения списка s представляются вырезками одного массива S. Для программы (7.10) необходимо кодировать начальное значение списка s и текущее, причем в начальный момент исполнения программы текущее значение устанавливается равным начальному. Начальное состояние – вырезка S[m..n], текущее – S[j..n]. Массив S и величины j, m, n должны быть определены в программе, причем j – переменная, а m и n могут быть константами.

```
type SEQR = array (real, M..N);
SEQR S;
int j = m;
```

Значения границ M и N должны быть достаточными, т. е. M ≤ m, n, j ≤ N. Операции со списком s кодируются следующим образом:

```
s = nil      →    j > n
s <> nil     →    j <= n
s.car  →          S[j]
s = s.cdr    →    j = j + 1
```

Применение трансформации кодирования списка s к программе (7.10) дает итоговую программу.

```
int j = m; c = 0; for (; j <= n; ) { c = c + S[j]; j = j + 1 }
```

7.6. Пример. Сортировка простыми вставками

Рассмотрим задачу сортировки простыми вставками [6]. На ее примере демонстрируются типовые методы построения предикатных программ, доказательства их корректности и получения эффективных программ. Представлен нетривиальный способ кодирования списков.

В алгоритме [6] сортируемый список s рассматривается в виде конкатенации u + b + v, где u и v – списки, а b – элемент, причем список u – сортированный. На очередном шаге работы алгоритма элемент b вставляется внутрь списка u таким образом, чтобы сохранить сортированность обновленного списка u. На следующем шаге в качестве элемента b рассматривается начальный элемент v, и процесс повторяется.

Элементы списка имеют произвольный тип T с линейным порядком "≤". Типы элементов и списков представлены описаниями:

type T{ ≤ : **axiom** ∀a. a ≤ a, **axiom** ∀a, b. a ≤ b & b ≤ a ⇒ a = b,
axiom ∀a, b, c. a ≤ b & b ≤ c ⇒ a <= c, **axiom** ∀a, b. a ≤ b ∨ b ≤ a };
type S = **list** (T);

Свойство *сортированности* списка s определяется предикатом:

SORT(s) ≅ ∀s1, s2 ∈ S. ∀a, b ∈ T. (s = s1 + a + b + s2 ⇒ a ≤ b)

Лемма 7.1. Список не более, чем из одного элемента – сортированный, т. е. SORT(nil) & ∀a. SORT([[a]]).

Лемма 7.2. length(s) ≤ 1 ⇔ s = nil ∨ ∃a. s = [[a]].

Лемма 7.3. length(s) ≤ 1 ⇒ SORT(s).

Допустим, список v является результатом сортировки исходного списка u. Списки u и v состоят из одной и той же совокупности элементов. Тогда список v можно получить из u применением серии перестановок элементов. Обозначим это свойство через PERM(u, v). Элементарную перестановку можно определить предикатом:

perm(u, v) ≅ ∃s1,s2,s3∈S. ∃a,b∈T. u = s1 + a + s2+ b + s3 & v = s1 + b + s2+ a + s3

Тогда свойство *перестановочности* PERM(u, v) можно определить следующим образом:

PERM(u, v) ≅ u = v ∨ ∃ w. perm(u, w) & PERM(w, v) .

Приведенное рекурсивное определение эквивалентно следующему нерекурсивному:

PERM(u, v) ≡ ∃n≥1. ∃w_1...w_n ∈ S. u = w_1 & v = w_n & ∀i=1..n-1. perm(w_i, w_{i+1}) .

Лемма 7.4. length(u) ≤ 1 & PERM(u, v) ⇒ u = v.

Доказательство. Пусть истинны length(u) ≤ 1 и PERM(u, v). Предикат perm(u, w) – ложный для любого списка w, поскольку в определении предиката u должна содержать не менее двух элементов. Поэтому ложна вторая альтернатива предиката PERM(u, v), а значит, истинна первая, т. е. u = v. □

Спецификацию задачи сортировки представим определением:

sort(S s**:** S s′) **post** SORT(s′) & PERM(s, s′);

Напомним, что штрих в имени s′ предполагает склеивание переменных s и s′ в реализации, см. разд. 6.3.

Сначала задачу sort следует свести к более общей задаче sort1, в которой s = u + b + v, где u и v – списки, а b – элемент, причем список u – сортированный. Спецификация задачи sort1 следующая:

sort1(S u, T b, S v**:** S r) **pre** u ≠ nil & SORT(u) **post** SORT(r) & PERM(u + b + v, r);

Лемма 7.5. Спецификация sort1 реализуема и однозначна, т. е. существует единственный список r, удовлетворяющий постусловию.

Возможность сведения задачи sort к sort1 обеспечивается следующей леммой.

Лемма 7.6. length(s) > 1 ⇒ s = s.car + s.cdr.car + s.cdr.cdr.

Дадим определение предиката sort и докажем его корректность.

```
sort(S s: S s')
{       if (length(s) ≤ 1)      s' = s
        else                    sort1(s.car, s.cdr.car, s.cdr.cdr: s')
} post SORT(s') & PERM(s, s');
```

Доказательство корректности. Поскольку спецификация реализуема, а тело предиката sort – однозначный оператор, то в соответствии с теоремой 2.1 тождества спецификации и программы достаточно доказать истинность тела предиката из постусловия. Пусть SORT(s') и PERM(s, s') истинны. Пусть истинно условие length(s) ≤ 1. Истинность оператора s' = s следует из леммы 7.4. Пусть истинно условие length(s) > 1. Докажем истинность вызова sort1. Прежде всего, истинны условия корректности для полей car и cdr, поскольку s и s.cdr отличны от nil. Далее, истинно предусловие для аргументов вызова sort1, т. е. истинны SORT(s.car) (по лемме 7.3) и s.car ≠ nil. По лемме 2.8. предикат sort1 тождественен постусловию. Поэтому достаточно доказать истинность формулы:

SORT(s') & PERM(s.car + s.cdr.car + s.cdr.cdr, s').

Ее истинность следует из леммы 7.6 и предусловия sort. □

Задача sort1 содержит независимую подзадачу pop_into вставки элемента b внутрь отсортированного списка u:

```
pop_into(S u, T b: S w) pre u ≠ nil & SORT(u) post SORT(w) & PERM(u + b, w);
```

Лемма 7.7. Спецификация pop_into реализуема и однозначна.

Дадим определение предиката sort1 и докажем его корректность.

```
sort1(S u, T b, S v: S r) pre u ≠ nil & SORT(u)
{       pop_into(u, b: S w);
        if (v = nil )           r = w
        else                    sort1(w, v.car, v.cdr: r)
} post SORT(r) & PERM(u + b + v, r);
```

Доказательство. Доказательство истинности тела предиката из предусловия и постусловия сводится к доказательству истинности условного оператора. Предусловие оператора pop_into(u, b: S w) истинно. Поэтому оператор полагается истинным при доказательстве истинности условного оператора. Мы можем заменить его постусловием SORT(w) & PERM(u + b, w). Переменная v является индуктивной. Используется отношение порядка: $v \sqsubseteq t \cong \exists q.\ q + v = t$. Тогда истинно $v.cdr \sqsubset v$. Минимальный элемент nil покрывается условием v = nil в условном операторе. Пусть истинно условие v = nil. Тогда из истинности PERM(u + b, r) и PERM(u + b, w) следует истинность оператора r = w. Ввиду истинности предусловия операций v.car и v.cdr, истинности предусловия рекурсивного вызова sort1 и истинности $v.cdr \sqsubset v$ в соответствии с индуктивным предположением можно заменить вызов sort1 постусловием SORT(r) & PERM(w + v, r). Истинность PERM(w + v, r) следует из истинности PERM(u + b, w) и PERM(u + b + v, r). □

Задачу pop_into необходимо свести к более общей задаче pop_into1, в которой u = y + a + z, где текущий элемент a разделяет списки y и z; при этом все элементы из z больше, чем b. Спецификация задачи pop_into1 следующая:

```
pop_into1(S y, z, T a, b: S w)      pre SORT(y + a + z) & (z ≠ nil ⇒ b ≤ z.car)
                                    post SORT(w) & PERM(y + a + b + z, w);
```

Лемма 7.8. Спецификация pop_into1 реализуема и однозначна.

Дадим определение предиката pop_into и докажем его корректность.

```
pop_into(S u, T b: S w) pre u ≠ nil & SORT(u)
{       pop_into1(prec(u), nil, last(u), b: w)
} post SORT(w) & PERM(u + b, w);
```

Доказательство. Нетрудно проверить истинность предусловия вызова pop_into1. Заменим вызов pop_into1 его постусловием SORT(w) & PERM(u + b, w) и обнаружим, что оно совпадает с постусловием pop_into, что доказывает его истинность. □

Определение pop_into1 реализуется разбором случаев. Докажем его корректность.

```
pop_into1(S y, z, T a, b: S w) pre SORT(y + a + z) & (z ≠ nil ⇒ b ≤ z.car)
{       if (a ≤ b)      w = y + a + b + z
        else   if (y = nil)    w = b + a + z
               else            pop_into1(prec(y), a + z, last(y), b: w)
} post SORT(w) & PERM(y + a + b + z, w);
```

Доказательство. Из предусловия и постусловия докажем истинность объемлющего условного оператора. Пусть истинно условие a ≤ b. Тогда список y + a + b + z является сортированным. Истинность оператора w = y + a + b + z следует из леммы 7.9.

Лемма 7.9. PERM(v, w) & SORT(v) & SORT(w) ⇒ v =w.

Пусть истинно a > b. Доказательство истинности внутреннего условного оператора проводится индукцией по переменной y. Отношение порядка несколько иное по сравнению с используемым в предикате sort1: y ⊑ t ≅ ∃q. y + q = t. Минимальный элемент nil покрывается условием y = nil в условном операторе. Пусть истинно y = nil. Тогда список b + a + z является сортированным и по лемме 7.9 истинен оператор w = b + a + z. Ввиду истинности предусловия операций prec(y) и last(y), истинности предусловия рекурсивного вызова pop_into1 и истинности prec(y) ⊏ y в соответствии с индуктивным предположением можно заменить вызов pop_into1 его постусловием SORT(w) & PERM(y + b + a + z, w). Истинность PERM(y + b + a + z, w) следует из истинности PERM(y + b + a + z, y + a + b + z). □

Применим к предикатной программе сортировки, составленной из определений предикатов sort, sort1, pop_into и pop_into1, систему трансформаций для получения эффективной императивной программы. На первом этапе реализуются склеивания переменных. Ниже приводится результат склеивания для первых трех предикатов; в pop_into1 склеиваний не производится. Перед определением предиката перечисляется список применяемых склеиваний. Отметим, что замена s ← r не является склеиванием – она делается для облегчения проведения дальнейшей подстановки определения.

```
s ← s'
sort(S s: S s)
{  if (length(s) ≤ 1)    s = s
   else                  sort1(s.car, s.cdr.car, s.cdr.cdr: s)
}
s ← r; u ← w;
sort1(S u, T b, S v: S s)
{  pop_into(u, b: S u);
   if (v = nil )         s = u
   else                  sort1(u, v.car, v.cdr: s)
}
u ← w;
pop_into(S u, T b: S u)
{  pop_into1(prec(u), nil, last(u), b: u) };
```

На втором этапе трансформаций заменим хвостовую рекурсию циклом в определениях sort1 и pop_into1. Раскроем групповые операторы присваивания и оформим циклы. При раскрытии группового оператора в pop_into1 возникает две коллизии, которые устраняются перестановками присваиваний.

```
sort1(S u, T b, S v: S s)
{       for (; ;) {
            pop_into(u, b: S u);
            if (v = nil )       {s = u; break}
            else                { b = v.car; v = v.cdr}
        }
}

pop_into1(S y, z, T a, b: S u)
{       for (; ;) {
                if (a ≤ b)      { u = y + a + b + z; break}
                else            if (y = nil)    { u = b + a + z; break}
                                else            {z = a + z; a = last(y); y = prec(y)}
        }
}
```

На третьем этапе подставим sort1 на место вызова в sort и pop_into1 в pop_into.

```
sort(S s: S s)
{       if (length(s) ≤ 1)      return
        S u =s.car; T b = s.cdr.car; S v = s.cdr.cdr;
        for (; ;) {
            pop_into(u, b: S u);
            if (v = nil )       {s = u; break}
            else                { b = v.car; v = v.cdr}
        }
}

pop_into(S u, T b: S u)
{       T a = last(u); S y = prec(u); S z = nil;
        for (; ;) {
                if (a ≤ b)      { u = y + a + b + z; break}
                else            if (y = nil)    { u = b + a + z; break}
                                else            {z = a + z; a = last(y); y = prec(y)}
    }
}
```

Далее, подставим определение pop_into на место вызова в sort.

```
sort(S s: S s)
{       if (length(s) ≤ 1)      return ;
        S u = s.car; T b = s.cdr.car; S v = s.cdr.cdr;
        for (; ;) {
            T a = last(u); S y = prec(u); S z = nil;
            for (; ;) {
                if (a ≤ b)      { u = y + a + b + z; break}
                else            if (y = nil)    { u = b + a + z; break}
                                else            {z = a + z; a = last(y); y = prec(y)}
             }
             if (v = nil )              {s = u; break}
             else                       { b = v.car; v = v.cdr}
        }
}
```

На четвертом этапе реализуется кодирование списков s, u, v, y и z вырезками одного массива. Особенность кодирования в том, что списки u и v находятся внутри s, а y и z – внутри u. Тип массива представим описанием:

type SEQ(**int** p, q) = **array** p..q **of** T;

Пусть список w кодируется вырезкой W[p..q]. Кодирование используемых в программе конструкций реализуется следующим образом:

S w → SEQ(p, q)
w = nil → p > q
length(w) → q – p + 1
b = w.car; w := w.cdr → b := W[p]; p := p + 1
a = last(w); w := prec(w) → a := W[q]; q := q – 1

Пусть список s кодируется вырезкой K[0..n].

int n; // номер последнего элемента списка
SEQ(0, n) K;

Пусть s = u + b + v. Здесь u и v являются частью списка s и кодируются вырезками массива K. Пусть позиция элемента b в массиве K есть i. Тогда

$$u \to K[0 \,..\, i-1];\ b \to K[\,i\,];\ v \to K[i+1 \,..\, n]\,. \qquad (7.11)$$

Отметим, что здесь один индекс i используется для представления двух соседних списков u и v. Особенность в том, что изменение индекса i определяет синхронное изменение этих списков. Аналогичным образом реализуется представление для u = y + a + b + z. В представлении используется другой индекс j.

$$y \to K[0..j-2];\ a \to K[j-1];\ b \to K[j];\ z \to K[j+1 \,..\, i]\,.$$

Позиция элемента b здесь иная, нежели в представлении (7.11). Другая особенность: позиция последнего элемента z совпадает с позицией b в (7.11). Это значит, что весь список z сдвинут на один элемент по сравнению с тем положением, которое имеет z в составе u в (7.11).

Приведем программу, полученную в результате кодирования списков вырезками массива. В ней описание с инициализацией **int** i = 1 используется как эквивалент двух конструкций S u = s.car и S v = s.cdr.cdr, которым соответствуют вырезки K[0..0] и K[2..n]. Аналогичным образом конструкция **int** j = i используется вместо S y = prec(u) и S z = nil, отображающихся в вырезки K[0..i – 2] и K[i + 1 .. i]. Кроме того, используется кодирование z = a + z → K[j] = a.

```
sort(S s: S s)
{  if (n ≤ 0)   return ;
   int i = 1; T b = K[1];
   for (; ;) {
      T a = K[i – 1]; int j = i;
      for (; ;) {
           if (a ≤ b)  { K[j] = b; break}
           else        if (j < 2)     { K[0] = b; K[1] = a; break}
                       else           {K[j] = a; a = K[j – 2]; j = j – 1 }
      }
      if (i >= n)      break
      else             { b = K[i + 1]; i = i + 1 }
   }
}
```

Наконец, приведем итоговую программу сортировки, отличающуюся от предыдущей деталями оформления циклов и упрощением условий.

```
type SEQ(int p, q) = array p..q of  T;
int n;              //номер последнего элемента списка
SEQ(0, n) K;
...
if (n ≤ 0) return ;
T b = K[1];
for (int i = 1; ;) {
    T a = K[i - 1];
    for (int j = i; ; j = j – 1) {
        if (a ≤ b)  { K[j] = b; break}
        else        if (j = 1)      { K[0] = b; K[1] = a; break }
                    else            { K[j] = a; a = K[j – 2] }
    }
    if (i = n)      break
    else            { i = i + 1; b = K[i] }
}
```

Приведенный пример построения программы сортировки простыми вставками в достаточной степени демонстрирует стиль и технологию предикатного программирования. Корректность предикатной программы строго доказывается применением системы правил, определенных в разд. 2 и 5. Эффективность и компактность итоговой программы не хуже написанной вручную. Отметим нетривиальность применяемого синхронного способа кодирования смежных списков с использованием общего индекса.

Можно отметить следующую неэффективность итоговой программы. Если после выполнения оператора `b = K[i]` обнаруживается, что `a ≤ b` для `a = K[i – 1]`, то оператор `K[j] = b` является избыточным. Данный недостаток легко устраняется другим алгоритмом, представленным ниже. Отметим, что предикатное программирование обладает гораздо большей гибкостью в сравнении с императивным и позволяет эффективно реализовать любое проектное решение.

```
pop_into(S u, T b: S w) pre u ≠ nil & SORT(u)
{       T a = last(u);
        if (a ≤ b)      w = u + b
        else            pop_into2(prec(u), a, b: w)
} post SORT(w) & PERM(u + b, w);

pop_into2(S y, z, T b: S w) pre SORT(y + z) & z ≠ nil & b ≤ z.car
{       if (y = nil)    w = b + z
        else   T c = last(y);
               if (c ≤ b)       w = y + b + z
               else             pop_into2(prec(y), c + z, b: w)
} post SORT(w) & PERM(y + b + z, w);
```

Императивная программа, полученная по этой версии предикатной программы, более эффективна, однако несколько сложнее и длиннее. Данный алгоритм используется в работе [7], однако представлен для массивов, а не списков. Использование массивов вместо списков делает алгоритм существенно сложнее. Намного сложнее становится доказательство корректности, поскольку программа представляется логикой второго порядка.

7.7. Гиперфункции

Пример 7.2. Допустим, для списка `s` целых чисел требуется извлечь второй элемент и присвоить переменной `e`. Список может содержать менее двух элементов. Логическая пере-

менная exist = **true**, если второй элемент существует. Вычисление переменных e и exist можно представить следующим определением:

```
elemTwo(list (int) s: int e, bool exist )                                   (7.12)
{     if (s = nil ∨ s.cdr = nil)     exist = false
      else                           { e = s.cdr.car || exist = true }
} post exist = (s ≠ nil & s.cdr ≠ nil) & (exist ⇒ e = s.cdr.car)
```

Предположим, имеется фрагмент программы с вызовом предиката elemTwo и последующими вызовами предикатов B и D, причем e используется среди аргументов B:

```
elemTwo(s: e, exist); if (exist) B(e...) else D(...)                        (7.13)
```

Подставим тело elemTwo на место вызова. Внесем оператор **if** (exist) A(e...) **else** (...) в обе альтернативы первого условного оператора:

```
if ( s = nil ∨ s.cdr = nil )                                                (7.14)
          { exist = false; if (exist) B(e...) else D(...) }
else      { { e = s.cdr.car || exist = true }; if (exist) B(e...) else D(...) }
```

Далее реализуется специализация, позволяющая заменить внесенный условный оператор одной из альтернатив:

```
if (s = nil ∨ s.cdr = nil)    { exist = false || D(...) }
else                          { { e = s.cdr.car || exist = true }; B(e...) }
```

Обнаруживается, что переменная exist становится ненужной. Устраняя ее, получим результат подстановки тела elemTwo на место вызова в более компактном виде:

```
if (s = nil ∨ s.cdr = nil) D(...) else { e = s.cdr.car; B(e...) }           (7.15)
```

В данном примере определение (7.12) выглядит неадекватным и громоздким. Адекватным можно считать лишь итоговый оператор (7.15). Другой недостаток: переменная e не определена для списков, содержащих менее двух элементов. Неудобные ситуации, аналогичные приведенной в примере 7.2, регулярно встречаются в процессе программирования.

Проблему, представленную в примере 7.2, рассмотрим в общем виде. Пусть имеется тройка Хоара P(x) { A(x: y, z) } Q(x, y, z), где x, y и z – непересекающиеся, возможно пустые, наборы переменных. Предикат Q(x, y, z) определяется следующим тождеством:

$$Q(x, y, z) \cong (C(x) \Rightarrow S(x, y)) \,\&\, (\neg C(x) \Rightarrow R(x, z)) \qquad (7.16)$$

Предикаты S(x, y) и R(x, z) можно рассматривать как функции S: x → y и R: x → z. В случае, когда условие C(x) истинно, предикат Q(x, y, z) совпадает с функцией S: x → y, а при ложном условии C(x) – с функцией R: x → z. Будем считать, что область определения каждой из функций имеет непустое пересечение с областью истинности предусловия P(x). Таким образом, Q(x, y, z) является результатом склеивания двух функций S: x → y и R: x → z. В предикате Q(x, y, z) не определено значение набора z, если C(x) истинно. Аналогично не определено значение y при ложном C(x).

Лемма 7.10. Если предикат S(x, y) реализуем в области P(x) & C(x), а R(x, z) – в области P(x) & ¬C(x), то спецификация предиката A(x: y, z) является реализуемой.

Лемма 7.11. Если предикат S(x, y) является однозначным в области P(x) & C(x), а R(x, z) – в области P(x) & ¬C(x), то спецификация предиката A(x: y, z) вляется однозначной.

Пусть вызов предиката A используется во фрагменте вида

```
A(x: y, z); if (C(x)) B(y: u) else D(z: u) .                                (7.17)
```

Вычисление C(x) может оказаться достаточно трудоемким. Очевидно, что значение C(x) можно вычислить внутри определения предиката A. Введем логическую переменную c и следующим образом инструментируем определение предиката A, превратив его в A′(x: c, y, z). Если некоторая ветвь тела предиката A завершается присваиванием набору y, то в конце этой ветви вставляется оператор c = **true**; если ветвь завершается вычислением z, то в конце ее вставляется c = **false**. Тогда фрагмент (7.17) можно заменить более эффективным:

A′(x: c, y, z); **if** (c) B(y**:** u) **else** D(z**:** u) . (7.18)

Пример 7.3. Необходимость использования предиката A′ вместо A рассмотрим на примере задачи решения системы линейных уравнений:

$$\mathrm{LinEq}(n, a, x, b) \cong \sum_{i=1}^{n} a_{ij}\, x_j, = b_j;\ j = 1..n.$$

Спецификацию задачи представим описаниями:

type VEC(**nat** k) = **array** (**real**, 1..k);
type MATRICE(**nat** k) = **array** (**real**, 1..k, 1..k);
Lin(**nat** n, MATRICE(n) a, VEC(n) b**:** VEC(n) x)
pre det(n, a) ≠ 0 [20]
post LinEq(n, a, x, b);

В предусловии функция det(n, a) обозначает детерминант матрицы. Истинное предусловие определяет невырожденность системы уравнений и гарантирует существование решения, т. е. реализуемость приведенной спецификации.

Иногда требуется решить более общую задачу:

Lin1(**nat** n, MATRICE(n) a, VEC(n) b**: bool** c; VEC(n) x)
post c = (det(n, a) ≠ 0) & (c ⇒ LinEq(n, a, x, b));

Предусловие предиката Lin1 по форме соответствует (7.16) с пустым предикатом R и S = LinEq. Переменная c вычисляет условие det(n, a) ≠ 0, аналогичное C(x) в (7.17). Использование Lin1 в программе реализуется в составе следующего фрагмента вида (7.18):

Lin1(n, a, b**:** c; x); **if** (c) B(x...) **else** D(...) .

В данном фрагменте вычисление c в Lin1 позволяет избежать повторного вычисления det(n, a).

Рассмотрим процесс подстановки определения A′ на место вызова во фрагменте (7.18) с последующим упрощением аналогично тому, как это делалось для фрагмента (7.13). Обнаруживается, что втягивание условного оператора внутрь подставленного тела A′ аналогично (7.14) с последующей специализацией может привести к появлению нескольких копий вызовов B(y...) и D(z...). Поэтому применим другое преобразование. Альтернативы условного оператора пометим метками 1 и 2. Получим оператор

if (c) 1: B(y**:** u) **else** 2: D(z**:** u) . (7.19)

Далее в подставленном теле A′ всякий оператор c = **true** заменим на **#**1, а c = **false** – на **#**2. Операторы **#**1 и **#**2 обозначают операторы перехода **goto** 1 и **goto** 2, соответственно. Например, применение данного преобразования для (7.13) дает следующий фрагмент:

if (s = nil ∨ s.cdr = nil) #1
else { e = s.cdr.car; #2 }**;**
if (exist) 1: B(e...) **else** 2: D(...)

Модифицированный фрагмент (7.18) запишем в виде

A″(x: y, z); **if** (c) 1: B(y**:** u) **else** 2: D(z**:** u) . (7.20)

Здесь A″ обозначает блок, полученный после подстановки определения A′ и замены c = **true** и c = **false** на **#**1 и **#**2. Поскольку в A″ нет переменной c, заменим оператор (7.19) последовательностью

1: B(y**:** u); **goto** 3; 2: D(z**:** u); 3: . (7.21)

Композицию вида (7.21) с двумя (или более) входами и слиянием ветвей, соответствующих разным входам, запишем в следующем виде:

case 1: B(y**:** u) **case** 2: D(z**:** u) . (7.22)

Каждый из операторов **case** 1: B(y**:** u) и **case** 2: D(z**:** u) называется *оператором продолжения ветви*, или, коротко, *оператор-ветвь*. Последовательность из одного или более операторов продолжения ветвей называется *обработчиком ветвей*. Обработчику ветвей пред-

[20] Ввиду возможной погрешности обычно используют условие det(a) > ε с константой ε, близкой к нулю.

шествует оператор – вызов гиперфункции или блок, содержащий операторы перехода. Если предшествующий оператор нормально завершается (не оператором перехода), то обработчик ветвей пропускается – исполнение продолжается с оператора, следующего за обработчиком.

Отметим, что оператор (7.19) эквивалентен

if (c) **#**1 **else #**2; **case** 1: B(y**:** u) **case** 2: D(z**:** u) . (7.23)

Отметим также, что блок A″(x: y, z) эквивалентен композиции

A′(x: c, y, z); **if** (c) **#**1 **else #**2 . (7.24)

В итоге, исходный фрагмент (7.17) эквивалентен следующему:

A″(x: y, z) **case** 1: B(y**:** u) **case** 2: D(z**:** u) . (7.25)

Исходный предикат A(x: y, z) будем записывать в виде A(x**:** y #1**:** z #2) и называть *гиперфункцией* с двумя *ветвями*. Метки 1 и 2 являются *метками ветвей*. *Определение гиперфункции* A(x**:** y #1**:** z #2) записывается следующим образом:

A(x**:** y #1**:** z #2) **pre** P(x); **pre** 1: C(x) (7.26)
{ A″(x: y, z) } **post** 1: S(x, y); **post** 2: R(x, z);

Здесь блок A″(x: y, z) является телом определения гиперфункции; P(x) – общим предусловием; C(x) – *предусловием первой ветви* гиперфункции; S(x, y) – *постусловием первой ветви*; R(x, z) – *постусловием второй ветви*. Фрагмент (7.17) записывается следующим образом:

A(x**:** y #1**:** z #2) **case** 1: B(y**:** u) **case** 2: D(z**:** u) . (7.27)

Для композиции (7.27) допускается более компактная запись:

A(x**:** y **:** z) **case** B(y**:** u) **case** D(z**:** u) . (7.28)

Вернемся к примеру 7.2 и дадим определение предиката elemTwo в виде гиперфункции:

```
elemTwo(list (int) s: int e: )
pre 1: s ≠ nil & s.cdr ≠ nil
{   if (s = nil ∨ s.cdr = nil)     #2
    else  { e = s.cdr.car          #1 }
} post 1: e=s.cdr.car;
```

Заголовок гиперфункции elemTwo(**list**(**int**) s**: int** e**:**) без указания меток ветвей подразумевает elemTwo(**list**(**int**) s**: int** e #1**:** #2). Отметим, что постусловие второй ветви отсутствует: оно считается тождественно истинным. Композиция фрагмента (7.13) записывается в новых обозначениях следующим образом:

elemTwo(s**:** e**:**) **case** B(e…) **case** D(…) .

Допустим, имеется определение предиката:

V(x, u) ≅ **pre** P_V(x) { A(x**:** y **:** z) **case** B(y**:** u) **case** D(z**:** u) } **post** Q_V(x, u); (7.29)

Гиперфункция A имеет определение (7.26). Предикаты B и D представлены тройками Хоара:

$\{P_B(y)\}$ B $\{Q_B(y, u)\}$, $\{P_D(z)\}$ D $\{Q_D(z, u)\}$,

Поскольку фрагмент (7.28) эквивалентен (7.17), определение (7.29) эквивалентно

V(x, u) ≅ **pre** P_V(x) {A(x: y, z); **if** (C(x)) B(y**:** u) **else** D(z**:** u)} **post** Q_V(x, u); (7.30)

Здесь предикат A определяется тройкой Хоара P(x) { A(x: y, z) } Q(x, y, z), где Q(x, y, z) представлена формулой (7.16). Определим предикат:

H(x, y, z, u) ≅ **pre** P_H(x, y, z){**if** (C(x)) B(y**:** u) **else** D(z**:** u)} **post** Q_H(x, y, z, u) .

Тогда получим следующие формулы для предусловия и постусловия предиката H:

$P_H(x, y, z) \cong (C(x) \Rightarrow P_B(y)) \,\&\, (\neg C(x) \Rightarrow P_D(z))$
$Q_H(x, y, z, u) \cong (C(x) \Rightarrow Q_B(y, u)) \,\&\, (\neg C(x) \Rightarrow Q_D(z, u))$.

Определение (7.30) эквивалентно следующему:

V(x, u) ≅ **pre** P_V(x) {A(x: y, z); H(x, y, z, u)} **post** Q_V(x, u); (7.31)

Построим правила RS1 и RS2 для суперпозиции в определении (7.31). Получим следующие правила.

Правило RS1′. $P_V(x) \vdash P(x) \,\&\, \forall y,z\,(Q(x, y, z) \Rightarrow P_H(x, y, z))$.

Правило RS2′. $P_V(x) \,\&\, \exists y,z\,(Q(x, y, z) \,\&\, Q_H(x, y, z, u)) \vdash Q_V(x, u)$.

Далее подставим формулы для Q, P_H и Q_H. Получим следующие правила.

Правило RS1″. $P_V(x) \vdash P(x) \;\&\; \forall y,z\,((C(x) \Rightarrow S(x, y)) \;\&\; (\neg C(x) \Rightarrow R(x, z)) \Rightarrow (C(x) \Rightarrow P_B(y)) \;\&\; (\neg C(x) \Rightarrow P_D(z)))$

Правило RS2″. $P_V(x) \;\&\; \exists y,z\,((C(x) \Rightarrow S(x, y)) \;\&\; (\neg C(x) \Rightarrow R(x, z)) \;\&\; (C(x) \Rightarrow Q_B(y, u)) \;\&\; (\neg C(x) \Rightarrow Q_D(z, u))) \vdash Q_V(x, u)$

Специализация по условию C(x) дает следующие правила.

Правило RS9. $P_V(x) \vdash P(x)$.

Правило RS10. $P_V(x) \;\&\; C(x) \vdash \forall y\,(S(x, y) \Rightarrow P_B(y))$.

Правило RS11. $P_V(x) \;\&\; \neg C(x) \vdash \forall z\,(R(x, z) \Rightarrow P_D(z))$.

Правило RS12. $P_V(x) \;\&\; C(x) \;\&\; \exists y\,(S(x, y) \;\&\; Q_B(y, u)) \vdash Q_V(x, u)$.

Правило RS13. $P_V(x) \;\&\; \neg C(x) \;\&\; \exists z\,(R(x, z) \;\&\; Q_D(z, u)) \vdash Q_V(x, u)$.

В итоге, справедлива следующая лемма.

Лемма 7.12. Пусть предусловие $P_V(x)$ истинно. Допустим, предикаты A, B и D корректны. Если правила RS9–RS13 истинны, то предикат V, представленный определением (7.29), является корректным.

Построим правила серии L для композиции (7.29), в которой используется гиперфункция. Для суперпозиции (7.31) построим правила LS1 и LS2.

Правило LS1′. $P_V(x) \vdash P(x)$.

Правило LS2′. $P_V(x) \;\&\; Q_V(x, u) \;\&\; Q(x, y, z) \vdash P_H(x, y, z) \;\&\; Q_H(x, y, z, u)$.

Для правила LS2′ подставим формулы для Q, P_H и Q_H. Получим правило LS2″.

Правило LS2″. $P_V(x) \;\&\; Q_V(x, u) \;\&\; (C(x) \Rightarrow S(x, y)) \;\&\; (\neg C(x) \Rightarrow R(x, z)) \vdash (C(x) \Rightarrow P_B(y)) \;\&\; (\neg C(x) \Rightarrow P_D(z)) \;\&\; (C(x) \Rightarrow Q_B(y, u)) \;\&\; (\neg C(x) \Rightarrow Q_D(z, u))$.

Специализация по условию C(x) дает следующие правила.

Правило LS10. $P_V(x) \vdash P(x)$.

Правило LS11. $P_V(x) \;\&\; Q_V(x, u) \;\&\; C(x) \;\&\; S(x, y) \vdash P_B(y) \;\&\; Q_B(y, u)$.

Правило LS12. $P_V(x) \;\&\; Q_V(x, u) \;\&\; \neg C(x) \;\&\; R(x, z) \vdash P_D(z) \;\&\; Q_D(z, u)$.

Лемма 7.13. Допустим, спецификация предиката V реализуема, предикаты A, B и D однозначны в области предусловий и корректны, а их спецификации однозначны. Если правила LS10, LS11 и LS12 истинны, то предикат V, представленный определением (7.29), является корректным.

Приведенные правила серий R и L для композиции (7.29), в которой используется гиперфункция, демонстрируют, что доказательство проводится раздельно для каждой ветви гиперфункции аналогично тому, как это делается для условного оператора. При этом доказательство для каждой из ветвей гиперфункции проводится как для суперпозиции.

Пример 7.4. Определим гиперфункцию Comp следующей спецификацией:

```
Comp(list (int) s: : int d, list (int) r)
pre 1: s = nil
post 2: d = s.car & r = s.cdr;
```

Построим определение предиката elemTwo из примера 7.2 через гиперфункцию Comp, не используя полей car, cdr и конструктора nil.

```
elemTwo(list (int) s: int e #1: #2)
pre 1: s ≠ nil & s.cdr ≠ nil
{   int e1; list (int) s1, s2;
    Comp(s: : e1, s1)
    case  #2
    case {
            Comp(s1: : e, s2)
            case #2
            case #1
    }
} post 1: e = s.cdr.car;
```

Если оператор продолжения ветви содержит только оператор перехода, то действует следующее правило: оператор-ветвь устраняется, а оператор перехода переносится в конец соответствующей ветви вызова гиперфункции. Определение гиперфункции elemTwo переписывается следующим образом:

```
elemTwo(list (int) s: int e #1: #2)
pre 1: s ≠ nil & s.cdr ≠ nil
{   Comp(s: #2: int e1, list (int) s1)
    case   Comp(s1: #2: e, list (int) s2 #1);
} post 1: e = s.cdr.car;
```

В данном определении elemTwo, кроме переноса указателей завершения, реализован перенос описаний локальных переменных e1, s1 и s2 в позиции их *определения*, т. е. места в программе, где им присваиваются значения.

Если обработчик ветвей состоит из единственного оператора продолжения ветви, а предшествующий оператор не имеет нормального завершения исполнения, то обработчик ветвей вырождается. В этом случае оператор продолжения ветви устраняется, а находящийся в нем оператор композируется с предыдущим оператором в виде оператора суперпозиции. Последнее определение elemTwo должно быть заменено следующим:

```
elemTwo(list (int) s: int e #1: #2)
pre 1: s ≠ nil & s.cdr ≠ nil
{   Comp(s: #2:  _, list (int) s1);
    Comp(s1: #2: e, _ #1);
} post 1: e = s.cdr.car;
```

Дополнительно в этом определении проведена замена результирующих вхождений локальных переменных e1 и s2 стандартным именем "_", означающим, что соответствующий результат игнорируется при исполнении.

Допустим, имеется условный оператор вида

```
if (C) {A; B} else {E; D} .                                   (7.32)
```

Определим гиперфункцию:

```
H(...: ...: ...) ≡ { if (C) {A; #1} else {E; #2} } .
```

Тогда условный оператор (7.32) эквивалентен следующей композиции:

```
H(...: ...: ...) case B case D .
```

Данное свойство демонстрирует гибкость аппарата гиперфункций. С помощью гиперфункций можно произвольно декомпозировать программу. Такое не реализуется традиционными средствами.

Стиль программирования с использованием гиперфункций проиллюстрируем на примере следующей задачи.

Пример 7.5. Сумма чисел в строке. Имеется строка литер, в которой встречаются числа в виде списка десятичных цифр. Числа разделяются произвольным набором литер, отличных от цифр. Подсчитать сумму чисел в строке.

Спецификацию задачи представим определением:

СуммаСтроки(**string** s**: nat** n) **post** P(s, n);

Предикат P(s, n) определяется рекурсивным образом.

цифра(**char** c) ≅ c ∈ {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

число(**string** s) ≅ s ≠ nil & ∀s1,s2∈**string** ∀c∈**char** (s = s1 + c + s2 ⇒ цифра(c))

val(**string** s**: nat** n) ≅ **pre** число(s) **post** n – значение числа s.

P(**string** s, **nat** n) ≅

(s = nil ⇒ n = 0) &

& (число(s) ⇒ n = val(s))

& (∀s1∈**string** ∀c∈**char**. ¬цифра(c) & s = c + s1 ⇒ P(s1, n)) &

& (∀s1,s2∈**string** ∀c∈**char**. ¬цифра(c) & число(s1) & s = s1 + c + s2 ⇒

∃m. P(s2, m) & n = val(s1) + m)

Напомним, что тип **string** есть **list**(**char**), а операция "+" обозначает конкатенацию строк.

Лемма 7.14. Спецификация предиката P реализуема.

Доказательство проводится индукцией по длине строки s. □

Так же, как и для программы УмнО (см. разд. 7.4), для построения алгоритма с хвостовой формой рекурсии необходимо обобщение исходной задачи введением накопителя для суммирования:

Сумма(**string** s, **nat** m**: nat** n) **post** ∃x. P(s, x) & n = m + x;

Ввиду истинности тождества СуммаСтроки(s**:** n) ≡ Сумма(s, 0**:** n) корректно следующее определение предиката:

СуммаСтроки(**string** s**: nat** n) ≡ {Сумма(s, 0**:** n)} **post** P(s, n);

Алгоритм решения данной задачи очевиден. Сначала в строке находится первая цифра. Далее вычисляется число, начинающееся с этой цифры. Для оставшейся части строки действие алгоритма повторяется. При нахождении первой цифры возможна ситуация, когда строка не содержит цифр. Данную ситуацию обозначим предикатом

мусор(**string** s) ≅ ∀s1,s2∈**string** ∀c∈**char** (s = s1 + c + s2 ⇒ ¬цифра(c)) .

В частности, предикат истинен для пустой строки. Определение первой цифры выражается гиперфункцией:

```
первыйЦифра(string s:  : char e, string r)
pre 1: мусор(s)
post 2: ∃w. мусор(w)  &  s = w + e + r  &  цифра(e);
```

Лемма 7.15. Спецификация гиперфункции первыйЦифра реализуема.

Отметим, что можно было бы использовать гиперфункцию вида первыйЦифра1(**string** s**: : string** r), где r начинается с первой цифры, однако в реализации это приведет к повторному доступу к одной и той же литере в строке. В практике типичным является решение, которое можно представить предикатом

первыйЦифра2(**string** s**: char** e, **string** r) .

Здесь отсутствие цифр в строке s кодируется некоторым значением e, отличным от цифры. Это типичный программистский трюк, усложняющий семантику переменной e и логику программы в целом. Вред подобных трюков обычно проявляется для сложных программ. Отметим, что решение в виде гиперфункции дает более короткую и эффективную программу.

Вторая часть алгоритма – выделение числа в строке и его вычисление – представляется предикатом:

```
числоВстроке(char e, string r: nat v, string t)
pre цифра(e)
post ∃w. конецЧисла(r, w, t) & v = val(e + w);
конецЧисла(r, w, t) ≅ r = w + t  & (w = nil ∨ число(w)) & (t = nil ∨ ¬цифра(t.car);
```

Лемма 7.16. Спецификация предиката числоВстроке реализуема.

Дадим определение предиката Сумма и докажем его корректность.

```
Сумма(string s, nat m : nat n)
{       перваяЦифра(s: : char e, string r)
        case  n = m
        case  {числоВстроке(e, r: nat v, string t); Сумма(t, m + v: n)}
}  post ∃x. P(s, x) & n = m + x
```

Доказательство. Докажем истинность правила LS11. Пусть истинны формулы: мусор(s), ∃x. P(s, x) & n = m + x. Необходимо доказать, что n = m, т. е. истинно P(s, 0). Истинность P(s, 0) доказывается индукцией по длине строки s. Докажем истинность правила LS12. Пусть истинны формулы:

$$\exists x.\ P(s, x)\ \&\ n = m + x,\quad \exists w.\ \text{мусор}(w)\ \&\ s = w + e + r\ \&\ \text{цифра}(e),\ \neg\text{мусор}(s)\ . \tag{7.33}$$

Необходимо доказать истинность второго оператора продолжения ветви, представляющего суперпозицию вызовов предикатов числоВстроке и Сумма. Поскольку истинно предусловие вызова числоВстроке, присоединим его постусловие к формулам (7.33) и докажем истинность рекурсивного вызова Сумма(t, m + v: n). Индуктивной переменной является первый аргумент. Используется следующее отношение порядка: $t \sqsubseteq s \cong \exists w.\ w + t = s$. Минимальный элемент nil покрывается условием мусор(s), которому соответствует первый оператор продолжения ветви: для s = nil доказательство корректности уже проведено.

Из второй формулы (7.33) следует, что s = q + e + r для некоторого q. Из постусловия числоВстроке следует, что r = w + t. Тогда s = q + e + w + t и $t \sqsubset s$. В соответствии с индукционным предположением вызов Сумма(t, m + v: n) эквивалентен его постусловию:

$$\exists y.\ P(t, y)\ \&\ n = m + v + y\ . \tag{7.34}$$

Докажем истинность (7.34). Пусть истинны P(s, x) и P(t, y). Достаточно доказать, что x = v + y. При этом истинны s = q + e + w + t, мусор(q) и $w = \text{nil} \vee$ число(w). Легко доказать, что истинно число(e + w). Тогда применима функция val(e + w). Из постусловия числоВстроке следует, что v = val(e + w). Индукцией по длине q доказывается истинность P((e + w) + t, x). Наконец, истинность x = v + y следует из определения предиката P. □

Дадим определение гиперфункции перваяЦифра и докажем его корректность.

```
перваяЦифра(string s:  : char e, string r) pre 1: мусор(s)
{       if (s = nil)              #1
        else {
                { char a = s.car || string u = s.cdr };
                if (цифра(a)) { r = u || e = a      #2}
                else перваяЦифра(u: #1 : e, r  #2)
        }
} post 2: ∃w. мусор(w) & s = w + e + r & цифра(e);
```

Доказательство. Пусть истинна спецификация гиперфункции, т. е. формула

$$\neg\text{мусор}(s) \Rightarrow \exists w.\ \text{мусор}(w)\ \&\ s = w + e + r\ \&\ \text{цифра}(e) \tag{7.35}$$

Докажем истинность тела гиперфункции. Пусть истинно условие s = nil. Тогда истинно предусловие первой ветви мусор(s), что подтверждает истинность оператора перехода #1. Пусть истинно $s \neq \text{nil}$. Докажем истинность блока после **else**. Истинны корректно использование полей car и cdr. Поэтому достаточно доказать истинность внутреннего условного оператора при a = s.car и u = s.cdr. Пусть истинно условие цифра(a). Поскольку s = a + u, то ложно мусор(s) и из (7.35) следует s = w + e + r & мусор(w) & цифра(e). Тогда w = nil, e = a и r = u. Поскольку истинен оператор перехода #2, то блок {r = u || e = a #2} является истинным. Пусть истинно условие ¬цифра(a). Докажем истинность рекурсивного вызова. Поскольку $u \sqsubset s$, а для s = nil корректность гиперфункции доказана, то рекурсивный вызов можно заменить постусловием:

$$\neg\text{мусор}(u) \Rightarrow \exists q.\ \text{мусор}(q)\ \&\ u = q + e + r\ \&\ \text{цифра}(e)\ . \tag{7.36}$$

Докажем истинность (7.36). Пусть истинна ¬мусор(u). Тогда истинны ¬мусор(s) и правая часть импликации (7.35). Поскольку s = a + u и ¬цифра(a), то w = a + q для некоторого q и истинно мусор(q). Это доказывает формулу (7.36). □

Чтобы получить хвостовую рекурсию для предиката числоВстроке, необходимо ввести накопитель для вычисления значения очередного числа. Определим предикат:

взятьЧисло(**string** r, **nat** p**: nat** v, **string** t) ≡
post ∃w. конецЧисла(r, w, t) & val1(w, p, v);
val(s**:** n) ≡ **pre** число(s) **post** val1(s, 0, n);
val1(w, p, n) ≡ (w = nil ⇒ n = p) & (w ≠ nil ⇒ val1(w.cdr, p ∗ 10 + valD(w.car), n) ,

где valD(e) – значение цифры e. Тогда корректно следующее определение предиката:

числоВстроке(**char** e, **string** r**: nat** v, **string** t)
pre цифра(e)
{ взятьЧисло(r, val(e)**:** v, t) }
post ∃w. конецЧисла(r, w, t) & v = val(e + w);

Дадим определение предиката взятьЧисло и докажем его корректность.

```
взятьЧисло(string r, nat p: nat v, string t)
{       if (r = nil)    {v = p || t = r}
        else
        {       {char b = r.car || string q = r.cdr};
                if (цифра(b)) взятьЧисло(q, p ∗ 10 + val(b): v, t)
                else {v = p || t = q}
        }
} post ∃w. конецЧисла(r, w, t) & val1(w, p, v);
```

Доказательство. Пусть истинно постусловие предиката взятьЧисло. Докажем истинность тела предиката. Пусть истинно условие r = nil. Тогда w = nil и t = nil. Далее, v = p. Это доказывает истинность оператора v = p || t = r. Пусть истинно условие r ≠ nil. Докажем истинность второй альтернативы внешнего условного оператора. Истинны условия корректности для полей car и cdr. Далее достаточно доказать истинность внутреннего условного оператора при b = r.car и q = r.cdr. Тогда r = b + q. Пусть истинно условие цифра(b). Докажем истинность рекурсивного вызова. Поскольку q ⊏ r, а для r = nil корректность предиката доказана, то рекурсивный вызов можно заменить его постусловием:

∃h. конецЧисла(q, h, t) & val1(h, p ∗ 10 + val(b), v) . (7.37)

Докажем истинность (7.37). Поскольку r = b + q и цифра(b), то w = b + h для некоторого h и истинно конецЧисла(q, h, t). Из определения val1 при r ≠ nil следует, что val1(w, p, v) ≡ val1(h, p ∗ 10 + val(b), v). В итоге, истинна формула (7.37). □

Определения предикатов СуммаСтроки, Сумма, перваяЦифра, числоВстроке и взятьЧисло образуют полную предикатную программу. Рассмотрим процесс ее трансформации для получения эффективной императивной программы. На первом этапе реализуются склеивания переменных. Полный перечень склеиваемых переменных представлен ниже.

Сумма:	n ← m;	s ← r, t;
перваяЦифра:	s ← r, u;	e ← a;
числоВстроке:	s ← r, t;	
взятьЧисло:	← r, q, t;	v ← p;

Замена склеиваемых переменных на внешнюю переменную s делается в определениях предикатов числоВстроке и взятьЧисло для упрощения последующей подстановки определений на место вызовов. Представим результат склеивания переменных.

```
СуммаСтроки(string s: nat n){Сумма(s, 0: n)};
Сумма(string s, nat n : nat n)
{       перваяЦифра(s: : char e, string s)
        case  n = n
        case  {числоВстроке(e, s: nat v, string s); Сумма(s, n + v: n)}
}
перваяЦифра(string s:  : char e, string s)
{       if (s = nil)            #1
        else {
                { e = s.car || s = s.cdr };
                if (цифра(e)) { s = s || e = e      #2}
                else перваяЦифра(s: #1 : e, s  #2)
        }
}
числоВстроке(char e, string s: nat v, string s)
{ взятьЧисло(s, val(e): v, s) }
взятьЧисло(string s, nat v: nat v, string s)
{       if (s = nil)    {v = v || s = s}
        else
        {       {char b = s.car || s = s.cdr};
                if (цифра(b)) взятьЧисло(s, v * 10 + val(b): v, s)
                else {v = v || s = s}
        }
};
```

На втором этапе трансформаций заменим хвостовую рекурсию циклом в определениях Сумма, перваяЦифра и взятьЧисло. Удаляются операторы вида n = n. Как следствие, удаляется оператор продолжения **case** n = n, что требует явного указания переходов от вызова гиперфункции. Раскроем групповые операторы присваивания и оформим циклы.

```
СуммаСтроки(string s: nat n){Сумма(s, 0: n)};
Сумма(string s, nat n : nat n)
{       for (; ;) {
                перваяЦифра(s: #M1 : char e, string s #2)
                case 2:       {числоВстроке(e, s: nat v, string s); n= n + v}
        }
        M1:
}
перваяЦифра(string s: #1 : char e, string s #2)
{       for (; ;) {
                if (s = nil)            #1
                else {
                        { e = s.car || s = s.cdr };
                        if (цифра(e))  #2
                        else {}
                }
        }
}
числоВстроке(char e, string s: nat v, string s)
{ взятьЧисло(s, val(e): v, s) }
```

```
взятьЧисло(string s, nat v: nat v, string s)
{       for (; ;) {
                if (s = nil)    break
                else
                {       {char b = s.car || s = s.cdr};
                        if (цифра(b)) v = v * 10 + val(b)
                        else break
                }
        }
};
```

Подставим определения предикатов взятьЧисло и перваяЦифра на место вызовов. Проведем очевидные упрощения.

```
Сумма(string s, nat n : nat n)
{       for (; ;) {
                for (; ;) {
                        if (s = nil)    #M1
                        { char e = s.car || s = s.cdr };
                        if (цифра(e))  #2
                }
                case 2:       {числоВстроке(e, s: nat v, string s); n= n + v}
        }
        M1:
}
числоВстроке(char e, string s: nat v, string s)
{       v = val(e);
        for (; ;) {
                if (s = nil)    break
                {char b = s.car || s = s.cdr};
                if (цифра(b)) v = v * 10 + val(b)
                else break
        }
};
```

Далее, поскольку внутренний цикл в теле предиката Сумма не имеет нормального выхода, можно заменить оператор #2 на **break** и убрать скобки оператора продолжения ветви. Подставим определение числоВстроке на место вызова, а затем определение предиката Сумма в тело СуммаСтроки. Получим:

```
СуммаСтроки(string s: nat n)                                    (7.38)
{       n = 0;
        for (; ;) {
                for (; ;) {
                        if (s = nil)    return
                        { char e = s.car || s = s.cdr };
                        if (цифра(e)) break
                }
                nat v = val(e);
                for (; ;) {
                        if (s = nil)    break
                        {char b = s.car || s = s.cdr};
                        if (цифра(b)) v = v * 10 + val(b)
                        else break
                }
                n = n + v
        }
}
```

На четвертом этапе применяется трансформация кодирования строки s вырезкой массива. При этом разные значения списка s представляются вырезками одного массива S. Для программы (7.38) необходимо кодировать начальное значение строки s и текущее, причем в начальный момент исполнения программы текущее значение устанавливается равным начальному. Начальное состояние – вырезка S[m..p], текущее – S[j..p]. Массив S и величины j, m, p должны быть определены в программе, причем j – переменная, а m и p могут быть константами.

```
type STR = array (char, M..N);
STR S;
int j = m;
```

Значения границ M и N должны быть достаточными, т. е. $M \le m, p, j \le N$. Операции со списком s кодируются следующим образом:

```
s = nil          j > p
s.car   →        S[j]
s = s.cdr        j = j + 1
```

Применение трансформации кодирования строки s к программе (7.38) дает итоговую программу на императивном расширении языка P.

```
type STR = array (char, M..N);                                    (7.39)
STR S;
nat n = 0;
for (int j = m; ;) {
        for (; ;) {
                if (j > p)      return
                char e = S[j]; j = j + 1;
                if (цифра(e)) break
        }
        nat v = val(e);
        for (;j <= p;) {
                char b = S[j]; j = j + 1;
                if (цифра(b)) v = v * 10 + val(b)
                else break
        }
        n = n + v
}
```

Можно обнаружить следующий недостаток программы (7.39): если строка s завершается цифрой, то проверка исчерпания строки реализуется дважды. Чтобы исправить этот недостаток, следует вместо предиката числоВстроке(e, r: v, t) использовать гиперфункцию числоВстроке1(e, r: v1: v2, t), в которой первая ветвь реализуется при исчерпании входной строки r.

Возможен другой более простой алгоритм. На очередном шаге вычисляется значение очередного числа в строке в виде предиката числоВстроке(s: v, t), где v – значение очередного числа, а t – остаток непустой строки s. Если первый элемент s не является цифрой, то v = 0. Реализация данного алгоритма дает более короткую программу, чем (7.39), однако менее эффективную. По сравнению с (7.39) лишними действиями являются операторы v = 0 и n = n + v для каждого символа, отличного от цифры.

7.8. Заключение

Технология предикатного программирования предоставляет аппарат для доказательства корректности предикатных программ. При наличии спецификации в виде предусловия и постусловия для каждого определяемого предиката корректность предикатной программы может быть доказана строго математически. Используемая техника доказательства базируется на системе правил доказательства корректности для конструкций языка P. Здесь демонстрировалась техника, соответствующая правилам серии L для однозначных программ и спецификаций. Это техника доказательства в стиле синтеза программы. Логика, применяемая в процессе доказательства, аналогична той, которую обычно использует программист при построении программы. Процесс доказательства хорошо структурирован и даже для большой программы состоит из небольших независимых частей, что предопределяет возможность успешной автоматизации доказательства корректности предикатных программ.

Язык предикатного программирования позволяет писать программы в естественной логике и при этом обладает гораздо большей выразительностью в сравнении с языками императивного программирования. На языке предикатного программирования можно записать любой алгоритм (для задач с предикатной спецификацией), который представим на императивном языке. Аппарат гиперфункций предоставляет новые возможности адекватного и эффективного представления алгоритма. Гиперфункциям нет аналогов в императивных языках. Эффективность в предикатном программировании обеспечивается применением серии оптимизирующих трансформаций. В результате получается императивная программа, которая по эффективности не хуже написанной вручную и почти всегда короче.

Список литература

1. Мир Лиспа. Методы и системы программирования: Пер. с англ. М.: Мир, 1990. Т. 2. 318 с.

2. *Ершов А. П.* Введение в теоретическое программирование. М.: Наука, 1977. 288 с.

3. *Поттосин И. В.* Направленные преобразования линейного участка // Языки и системы программирования. Новосибирск, 1981. С. 14–28. (Сб./ВЦ СО АН СССР).

4. *Bacon D., Graham S., Sharp O.* Compiler transformations for high-performance computing // ACM Computing surveys. 1994. Vol. 26. No. 4. P. 345–420.

5. *Петров Э. Ю.* Склеивание переменных в предикатной программе // Методы предикатного программирования. Новосибирск, 2003. С. 48–61.

6. *Кнут Д.* Искусство программирования для ЭВМ. Сортировка и поиск: Пер. с англ. М.: Мир, 1978. Т.3. 843 с.

7. *Шелехов В. И., Карнаухов Н. С.* Демонстрация технологии предикатного программирования на задаче сортировки простыми вставками // Методы предикатного программирования. Новосибирск, 2003. С. 16–21.