СИСТЕМОЛОГИЯ И МЕТОДОЛОГИЧЕСКИЕ ПРОБЛЕМЫ
ИНФОРМАЦИОННО-ЛОГИЧЕСКИХ СИСТЕМ
(Вычислительные системы)
1990 год Выпуск 135

УДК 519.685

К ОПЕРАЦИОННОЙ ИНТЕРПРЕТАЦИИ Σ^+-ПРОГРАММ. ЯЗЫК r-ПРОЦЕДУР

С.В. Котов

В в е д е н и е

Язык Σ^+-программ [3] является основным примером формализма, предназначенного для апробации концепции семантического программирования [1,2,6]. Наиболее значительная, с точки зрения данной работы, идея этой концепции состоит в том, что все логические рассуждения и поддерживающие их вычисления можно проводить с привлечением знаний о конкретной заданной формальной модели. В отличие от других стилей программирования, где модели используются для теоретического определения смысла языковых конструкций и/или обоснования правильности программ, семантическое программирование предлагает использовать (конструктивно описанную) модель непосредственно для решения задач. Разница в подходах особенно хорошо видна в сравнении с логическим программированием [5,8]. Логическое программирование в смысле ПРОЛОГа позволяет проверить выводимость некоторого факта из данного набора аксиом посредством применения стандартных правил вывода. Семантическое программирование предлагает определить истинность утверждения, проверяя возможные интерпретации утверждения в зафиксированной пользователем модели. Для этого могут использоваться и системы логического вывода (причем раз-

личные), и непосредственный перебор по элементам носителя модели.

Синтаксис языка Σ^+-программ и его денотационная семантика определены в [3], в дальнейшем предполагается, что читателю известно ее содержание. Там же описана и формальная операционная семантика языка построением дерева, отражающего процесс сведения поставленной задачи к набору подзадач и решение последних, если это оказывается возможным. С практической точки зрения подобное описание не устраивает нас по ряду причин:

- избыточность используемых конструкций, поскольку очевидно, что практическое получение решения задачи требует значительно меньших построений, чем предлагает данная семантика;

- теоретическая направленность языка описания стратегий решения; он, во-первых, не затрагивает ряда важных для реали - зации аспектов практического решения, во-вторых, цели его введения, а следовательно, и результаты далеки от реального программирования.

Данная работа представляет собой первый шаг в описании другого подхода к построению операционной семантики языка Σ^+-программ. Фактически предлагается модель многошаговой трансляции и исполнения этого языка на ЭВМ, ориентированная на практические приложения. Эта модель включает в себя предложения по организации и представлению структур данных по промежуточным этапам трансляции, по алгоритмам трансляции и т.д., поэтому мы говорим о модели системы семантического программирования на базе языка Σ^+-программ. Одновременно на работу можно и необхо - димо смотреть как на описание операционной семантики языка Σ^+-программ, так как конечным результатом является модель реали - зации языка. Однако, ввиду многообразия способов определения истинности утверждения в формальной модели, мы не задаем опе - рационную семантику общепринятым сегодня способом - описанием системы переходов. Наша схема трансляции будет построена таким

образом, что допустит конкретизацию практически до любого способа определения истинности.

Ключевая идея трансляции состоит в введении промежуточного формализма r-процедур и модели представления данных, удобной для описания императивной семантики этого языка. Разработка языка преследовала, как минимум,две цели.

Во-первых, заполнить вакуум между декларативными языками высокого уровня (в даной работе это язык Σ^+-программ) и императивными языками, близкими по духу к распространенным архитектурам ЭВМ. Для достижения этой цели в конструкции языка r-процедур синтезированы идеи двух способов описания задач: в основе декларативного описания лежат Σ^+-формулы, чисто императивный способ описания базируется на конструкторах процедур.Чтобы связать эти "несовместимые" способы,наш язык обладает двумя взаимоувязанными семантиками: императивной и декларативной.

Во-вторых, предложить простое, но достаточно мощное формальное средство для исследования различных аспектов "исполнения" Σ^+-программ, т.е. извлечения знаний из конструктивной модели, описанной в логико-математических терминах. Язык r-процедур предельно прост, он содержит лишь выражения четырех видов: 1) введение новой динамической переменной; 2) поиск ответа на вопрос, который поставлен логической формулой; 3) последовательность, в определенном смысле, процедур и 4) итерацию процедур типа цикла "for". В [4] показано, что если в п.2 ограничиться формулами минимальной сложности, то язык остается достаточным средством для достижения указанных целей.

Поскольку мы видим конечной целью трансляции Σ^+-программ определение истинности логических утверждений, вполне естественно организовать язык в виде процедурного расширения языка Σ^+-формул. Пример такого расширения можно найти в [6], однако цели указанной работы далеки от наших, и поэтому мы предлагаем свое процедурное расширение. Основные принципы его таковы:

а) каждая Σ^+-формула является r-процедурой, однако все вхождения логических переменных заменяются на порты, связанные с динамическими переменными;

б) вводятся необходимые, а возможно, и просто полезные, дополнительные процедурные примитивы;

в) вводятся "императивные" конструкторы сложных процедур.

На наш взгляд, перспектива применения этого языка заключается, во-первых, в возможности сочетать алгоритмические и декларативные спецификации задач и тем самым позволить использовать различные стили программирования в рамках решения одной задачи и, во-вторых, в возможности формализовать трансляцию декларативных языков в императивные.

В первую очередь необходимо рассмотреть возможные трактовки логико-математических понятий "предикат" и "функция" в языке (системе) программирования. Понятия "функция" и "предикат" задействованы в языке Σ^+-программ, и, следовательно, они должны быть некоторым образом представлены в промежуточных формализмах схемы трансляции. Нам кажется полезным выбрать одно из этих двух понятий как основное, а затем определить способы его реализации.

Функции присутствуют почти во всех языках программирования, в том числе в ряде языков, например, основанных на ламбда-нотации; они (функции) являются основным исходным понятием. Функциональный стиль программирования хорошо известен и исследован как с теоретической, так и с практической стороны.

Гораздо менее популярными в программировании являются предикаты. Это связано в первую очередь с тем, что данное математическое понятие изначально лишено императивного смысла. Предикаты появились в языках программирования в контексте специальных средств обработки знаний, заданных в виде отношений. Основными примерами являются языки логического программирования (ПРОЛОГ и др.) с методом линейной гиперрезолюции, запросов ре-

ляционных баз данных, параллельного (concurrent) логического программирования.

Для эффективной и единообразной работы с объектами языка нам удобно остановиться на одном из двух обсуждаемых понятий, как основном, а второе выразить через первое. Таким образом, предлагается свести наш язык или к чисто функциональной, или к чисто предикатной методологии.

Очевиден способ кодирования предиката булевозначной функцией. Однако он обладает существенным недостатком: извлечь некоторую нетривиальную информацию, например, как это можно сделать запросом (goal) ПРОЛОГа, из такой функции очень сложно.

Способ сведения функции к предикату естественно основывать на введении предиката равенства. В логическом программировании это серьезно усложняет алгоритм вывода; логическое программирование с равенством является, по сути, самостоятельной проблемной областью. Однако работа с равенством сложна именно для метода резолюции, который работает в термальной модели. С точки зрения семантического программирования равенство ничем не отличается от любого другого предиката, более того, оно явно присутствует в синтаксисе Σ^+-программ. (Кстати, функции в языке Σ^+-программ имеют явно предикатную сущность, так как синтаксис Σ^+-определений функции никак не гарантирует функциональность в смысле "не более одного образа для элемента области определения".)

Наконец, хотелось бы также упомянуть одно преимущество предикатного подхода. Оно заключается в потенциальном разнообразии семантик, что хорошо видно на примере логического программирования (см. [9]).

Таким образом, мы пришли к решению считать предикат основным понятием нашей системы программирования. Открытым остается вопрос о возможных императивных смыслах предиката.

Языки логического программирования, если отвлечься от возможностей механизма унификации, предлагают нам до 2^n различных процедурных прочтений n-арного предиката. Например, конструктивно описанное бинарное отношение R допускает нахождение ответа на такие запросы, как:

1) "для данного a найти все y такие, что $R(a,y)$ верно";

2) "для данного b найти все x такие, что $R(x,b)$ верно";

3) "найти все пары $\langle x,y \rangle$ такие, что $R(x,y)$ верно";

4) "для данных a и b определить, верно ли $R(a,b)$".

Назовем такого рода прочтения процедурами (или процедурными интерпретациями предиката). Из приведенного примера видно, что эти процедуры не являются обычными в программистском смысле, так как:

а) для одного заданного набора значений входных параметров может существовать (и должно выдаваться "на выходе") множество результатов (в том числе и пустое);

б) если процедура имеет m выходных параметров, то результат есть множество m-ок, а не m-ка множеств;

в) в последнем примере важно заметить, что под процедурой, интерпретирующей отношение R в контексте "все параметры входные" понимается объект с тремя (!) параметрами: два входных, по числу параметров отношения, и один выходной, выдающий ответ типа истина/ложь.

Далее, полезно отметить, что конкретное отношение R может иметь меньше процедурных интерпретаций, чем 2^n. Например, пусть $\{P_i(y)\}$, $i \in I$, есть семейство одноместных предикатов, причем множество индексов не рекурсивно-перечислимо. Тогда $R(i,y) = P_i(y)$ есть бинарное отношение, не допускающее запрос вида 2. Другой пример можно найти в языках парал-

лельного (concurrent) логического программирования, где мощным средством контроля за исполнением программ является требование использования некоторых параметров предикатов или только в качестве входных, или только в качестве выходных. Наконец, ПРОЛОГ предлагает записывать процедуры ввода-вывода, контроля исполнения, изменения программы и т.д. как предикаты, однако ясно, что такие предикаты имеют только одно процедурное прочтение.

Все эти рассуждения привели нас к модели вычислений, изложенной в § 1, и языку процедур, в котором представимы предикаты, подробное определение которого составляет § 2. Заключение посвящено краткому описанию собственно многошаговой трансляции Σ^+-программ, а также перспективам развития поднятой в работе темы.

§ 1. Модель вычислений

Перейдем к изложению идей нашей модели вычислений и, в первую очередь, решения вопроса об операционной трактовке понятия "предикат". Объект, соответствующий в нашей системе предикату, мы уже условились называть процедурой. Процедура обладает арностью и ограничениями на параметры (такими, как типы), унаследованными от предиката.

Определим n-арную процедуру как объект, общающийся с "остальным миром" через $n+1$ порт. Точнее, через эти порты процедура связана с динамическими переменными. "Динамичность" переменной проявляется в ее способности менять свое значение, в отличие от переменных в логическом смысле. Порт характеризуется количеством связанных с ним переменных (при этом возможны два варианта - одна или две переменные), а также направлением передачи информации между процедурой и этими переменными. Порты могут быть одного из шести видов:

1) нефиксированный,

2) нефиксированный/входной,

3) нефиксированный/выходной,

4) входной,

5) входной/выходной,

6) выходной.

Порты видов, записанных через "/", связаны с двумя переменными, а остальные - с одной. Слова "нефиксированный","входной" и "выходной" определяют направление передачи данных. Один порт процедуры ((n+1)-й) всегда имеет шестой вид, т.е. он выходной и связан с выделенной в системе переменной TruthValue.

Переменные,с которыми связана через порты процедура P , разбиты на три группы:

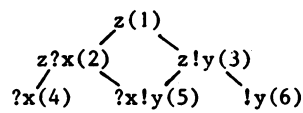
(z∈) Unf(P) - множество переменных с "нефиксирован - ной" ролью; их основное свойство заключается в том, что вид их взаимодействия с процедурой может быть доопределен до более конкретного;

(y∈) Out(P) - множество "выходных" переменных,принимающих результаты работы P; в случае, если это множество пусто для данного P, результат работы процедуры попадает в переменную TruthValue через (n+1)-й порт;

(x∈) In(P) - множество "входных" переменных,значения которых P использует для своей работы.

Кроме связи через порты, процедура, как и логическая формула,может иметь внешние переменные параметры , которые собраны во множество Ext(P). Мы считаем,что внешние переменные всегда означиваются до начала работы процедуры.

Для шести видов портов мы используем следующие обозначения (x,y,z - переменные, цифры в скобках - номер вида):



Порт вида, расположенного на приведеной схеме выше,может, в силу присутствия в нем нефиксированной переменной, перейти в порт вида, расположенного ниже и представляющего собой как бы частный случай. В языке r-процедур, который будет описан в следующем параграфе, переход к частному случаю может осуществляться простым синтаксическим преобразованием представляющего процедуру выражения. Это позволит нам в перспективе достаточно эффективно переходить от предиката к частным случаям его использования посредством механического преобразования.

Остановимся подробнее на динамических переменных. В силу того, что они хранят результаты работы процедур, интерпретирующих предикаты, их функции значительно сложнее, чем функции переменных в обычных языках программирования. Прототипом последних является ячейка физического устройства памяти. Вводимые же нами динамические переменные могут:

а) иметь одновременно множество значений, которое накапливается постепенно;

б) быть связаны между собой, если они хранят результаты работы одной и той же процедуры: дело в том, что предикат, m параметров которого являются искомыми (выходными), должен образовывать результат в виде множества m-ок значений, а не m-ки множеств значений.

Таким образом, динамическую переменную удобно представлять себе как некоторое устройство (канал), принимающий данные от нескольких процедур и передающий данные также нескольким процедурам. Причем его нельзя считать очередью, так как каждое хранящееся значение может быть считано из него несколько раз. Кроме того, необходимо учитывать возможные смысловые связи между переменными (п."б").

Из всего сказанного следует, что для работы с динамическими переменными необходимо разработать специальный формализм и в дальнейшем соответствующее программное обеспечение. Ниже

предложена теоретическая модель представления данных. Основные ее положения таковы:

- динамической переменной в каждый момент времени ставится в соответствие не один элемент носителя модели, а множество (возможно, пустое) элементов; на данном этапе мы хотим подчеркнуть, что это множество неупорядочено, т.е. реальный порядок хранения данных для нас несуществен и не имеет влияния на нашу модель вычислений;

- некоторые динамические переменные могут находиться в отношении синхронизованности, отмечающем смысловую связанность соответствующих им значений;

- совокупность всех динамических переменных, соответствующих им значений и отношения синхронизованности на переменных фиксирует состояние памяти системы.

1.1. Обозначения. Предполагается, что мы работаем в многосортной модели $\mathcal{M}$, множество сортов которой, обозначаемое Sort, определяет множество типов данных Type = $\{\tau \mid \tau \subseteq \mathrm{Sort}\}$. В дальнейшем мы нередко будем "забывать" о типах, предполагая, что необходимая коррекция "бестипового" изложения в "типовое" представляет лишь технические осложнения.

Пусть M_s обозначает носитель для сорта $s \in \mathrm{Sort}$, а $M_\tau = \bigcup_{s \in \tau} M_s$ - носитель для типа $\tau \in \mathrm{Type}$. Весь многосортный носитель модели $\mathcal{M}$ удобно представить как $M = \bigcup_{s \in \mathrm{Sort}} M_s$.

Напомним, что один из сортов обязательно есть list. Носитель M_{list} состоит из всех наследственно-конечных списков с праэлементами из носителей других сортов.

Пусть Var есть множество всех (логических и динамических) переменных. Изначально выделим одну переменную TruthValue $\in$ $\in$ Var; ее носитель будет состоять из одного элемента true.

Каждой переменной приписан тип из Type, его можно задать отображением $\tau: Var \to Type$. Предполагается, что в нашем распоряжении находится достаточно переменных любого типа и носители любого типа есть по крайней мере рекурсивно-перечислимые множества, что позволит нам считать, что существуют эффективные процедуры генерации элементов носителей. Далее мы предполагаем, что нам заданы фиксированные $Sort$, M_s $(s \in Sort)$, Var.

Полезными будут следующие обозначения. Для $V \subseteq Var$ обозначим $M(V) = \prod_{x \in V} M_{\tau(x)}$, где $\prod$ обозначает декартово произведение. Фактически $M(V)$ есть множество всех отображений a из V в M, таких, что $a(x) \in M_{\tau(x)}$. В случае, если V конечно, элементы $M(V)$, как обычно, будем называть n-ками.

Для $a \in M(V)$ и $W \subseteq V$ мы пишем $a|W$ для сужения a на W. Ясно, что $a|W \in M(W)$. Для произвольных $V, W \subseteq Var$, n-ки $a \in M(V)$ и $b \in M(W)$ мы называем согласованными и обозначаем это отношение $a \sim b$, если $a|(V \cap W) = b|(V \cap W)$. Для согласованных a и b обозначим через (a,b) такой элемент $M(V \cap W)$, что $(a,b)|V = a$ и $(a,b)|W = b$.

Операция "$|$" естественно расширяется на множества: если $A \subseteq M(V)$ и $W \subseteq V$, то определим $A|W$ как множество $\{ a|W \mid a \in A \}$. Ясно, что $A|W \subseteq M(W)$.

1.2. Модель представления данных. Пусть ϑ есть некоторое отношение эквивалентности на множестве переменных Var. Обозначим через $Var/_\vartheta$ фактор-множество, состоящее из классов эквивалентности $[x]_\vartheta = \{y \in Var \mid x \vartheta y\}$. Пусть отображение $\nu: Var/_\vartheta \to \bigcup_{V \in Var} M(V)$ таково, что каждому $[x]_\vartheta$ ставится в соответствие некоторое подмножество $M([x]_\vartheta)$.

ОПРЕДЕЛЕНИЕ 1. Пару $\langle \vartheta, \nu \rangle = I$, удовлетворяющую указанным выше требованиям, назовем состоянием динамических переменных. Переменные x и y , принадлежащие одному классу эквивалентности, т.е. такие, что $x \vartheta y$, назовем синхронизованными в состоянии I.

В дальнейшем мы будем использовать следующие обозначения:

$$I[x] = \nu([x]_\vartheta),$$
$$I/x/ = I[x] | \{x\},$$
$$[x]_I = [x]_\vartheta.$$

Обозначим через $\{x \to a\}$ состояние, интерпретирующее переменную $x \in Var$ элементом a, а все остальные переменные из Var - пустыми множествами элементов. Запись $I\backslash a$, где I - состояние и a - некоторая n-ка с $Dom(a) = W$, обозначает состояние, у которого $\vartheta = \vartheta_I \cup (W \times W)$. Другими словами, запись $I\backslash a$ означает, что все переменные, означиваемые через a, синхронизованы и $I\backslash a[x] = I[x]$, если $x \notin W$, и $I\backslash a[x] = \{a\}$, если $x \in W$.

"Физический" смысл введенных понятий уже был описан. Напомним, однако, что синхронизованность переменных $x, y, \ldots$ $(x \vartheta y \vartheta \ldots)$ означает их смысловую связанность, т.е. они хранят результаты работы некоторой процедуры и эти данные следует рассматривать как пары, тройки и т.д.

ОПРЕДЕЛЕНИЕ 2. Для данного $W \subseteq Var$ поток данных через W в состоянии I есть $S_W(I)$ - подмножество $M(W)$, задаваемое формулой:

$$\forall x \in W \; S_W(I) | (W \cap [x]_I) = I[x] | (W \cap [x]_I).$$

Очевидно, что $S_W(I)$ определяется приведенным условием однозначно. Фактически поток данных есть множество всех m-ок значений, которые поставлены в соответствие указанной m-ке переменных W в состоянии I.

ПРИМЕР. Пусть $x,y,z \in Var$, и $\vartheta_I = \{\langle x,z\rangle, \langle z,x\rangle, \langle x,x\rangle, \langle y,y\rangle, \langle z,z\rangle, \ldots\}$, $I[x] = I[z] = \{\langle 1,2\rangle, \langle 3,4\rangle\}$, $I[y] = \{5,6,7\}$. Тогда

$$S_{\langle x\rangle}(I) = \{1,3\},$$

$$S_{\langle y,z\rangle}(I) = \{\langle 2,5\rangle, \langle 2,6\rangle, \langle 2,7\rangle, \langle 4,5\rangle, \langle 4,6\rangle, \langle 4,7\rangle\},$$

$$S_{\langle x,y,z\rangle}(I) = \{\langle 1,2,5\rangle, \langle 1,2,6\rangle, \langle 1,2,7\rangle, \langle 3,4,5\rangle, \langle 3,4,6\rangle, \langle 3,4,7\rangle\}.$$

Пусть St - класс всех состояний для зафиксированного нами множества Var. При моделировании памяти реальной системы программирования, естественно сосредоточиться на случаях, когда только конечному подмножеству Var ставятся в соответствие непустые множества значений. Отождествим состояния, одинаково интерпретирующие это подмножество Var и различающиеся отношением синхронизованности на переменных, интерпретируемых пустым множеством значений. Для этого рассмотрим бинарное отношение $\approx$ на St, задаваемое правилом:

$$I \approx J \leftrightarrow \forall z \in Var\ ((I[z] \neq \emptyset \rightarrow [z]_I = [z]_J\ \&\ I[z] = J[z])\ \&\ (I[z] = \emptyset \rightarrow J[z] = \emptyset)).$$

УТВЕРЖДЕНИЕ 1. *Отношение "$\approx$" есть отношение эквивалентности.*

УТВЕРЖДЕНИЕ 2. *Пусть* $I \approx J$, *тогда* $\forall W \subseteq Var$ *имеем* $S_W(I) = S_W(J)$.

Обозначим через States фактор-множество $St/_{\approx}$. В дальнейшем будет использоваться это множество: его элементы будут представляться любым состоянием (т.е. элементом из St), принадлежащим соответствующему классу эквивалентности по "$\approx$". (Если $I \in St$, то мы будем писать $I \in$ States, имея в виду класс эквивалентности по $\approx$, содержащий I.)

Введем теперь частичный порядок на множестве States, интуитивно соответствующий отношению "I содержит меньше информации, чем J".

ОПРЕДЕЛЕНИЕ 3. Для $I,J \in States$ полагаем

$$I \subseteq J \Leftrightarrow \vartheta_J \subseteq \vartheta_I \;\&\; \forall x \in Var\ I[x]|[x]_J \subseteq J[x].$$

Обозначив точную верхнюю и нижнюю грани для пары элементов $\cup$ и $\cap$, имеем

УТВЕРЖДЕНИЕ 3. *Для любых* $I,J \in States$ *существует* $K \equiv I \cap J$, *определяемая формулой* $\vartheta_K = \vartheta_I \vee \vartheta_J$ & & $K[x] = (S_{[x]_K}(I) \cap S_{[x]_K}(J))$.

УТВЕРЖДЕНИЕ 4. *Для любых* $I,J \in States$ *существует* $K \equiv I \cup J$, *определяемая формулой* $\vartheta_K = \vartheta_I \wedge \vartheta_J$ & & $K[x] = (S_{[x]_K}(I) \cup S_{[x]_K}(J))$.

УТВЕРЖДЕНИЕ 5. *Для любых* $I,J,K \in States$ *верно*

$$K = I \cup J \Leftrightarrow \vartheta_K = \vartheta_I \wedge \vartheta_J \;\&\; K[x] =$$
$$= I[x]|[x]_K \cup J[x]|[x]_K = (I[x] \cup J[x])|[x]_K.$$

Здесь "$\vee$" и "$\wedge$" обозначают точные верхнюю и нижнюю грани эквивалентностей в решетке эквивалентностей над множеством Var: "$\wedge$" есть в точности теоретико-множественное пересечение эквивалентностей, а "$\vee$" есть транзитивное замыкание их теоретико-множественного объединения.

Из существования точных граней следует

УТВЕРЖДЕНИЕ 6. *Частично упорядоченное множество* $\langle States, \subseteq \rangle$ *есть решетка.*

Можно также показать, что верно

УТВЕРЖДЕНИЕ 7. *Частично упорядоченное множество* $\langle States, \subseteq \rangle$ *есть полная решетка.*

ОПРЕДЕЛЕНИЕ 4. Наименьший элемент $\langle States, \subseteq \rangle$ назовем пустым состоянием и обозначим через 0.

§ 2. Язык r-процедур

Перейдем к определению языка r-процедур. Для формального описания класса выражений, представляющих собой r-процедуры, мы должны поставить в соответствие каждой процедуре P четыре конечных множества переменных: $Ext(P)$, $Unf(P)$, $Out(P)$, $In(P)$. Содержательный смысл переменных из этих множеств описан в предыдущем параграфе. Объединение всех четырех множеств для данной P будем обозначать через $Var(P)$.

Обозначим через $Proc(\sigma)$ наименьшее множество выражений, удовлетворяющих нижеприведенным правилам RS1-RS5. Это множество образует класс r-процедур сигнатуры σ.

Одновременно эти правила задают понятие подпроцедуры (Sub) и атомарной подпроцедуры $(AtSub)$, а также для каждой процедуры из $Proc(\sigma)$ указанные выше четыре множества переменных. Кроме того, с каждым правилом RS1-RS5 параллельно дается определение императивной семантики выражений, разъясняющих смысл соответствующей синтаксической конструкции r-процедур. Ниже, как правило, мы будем писать просто Proc, считая параметр σ фиксированным. Прежде чем дать определения синтаксиса, изложим общие положения императивной семантики.

В § 1 мы ввели понятие состояния (вычислительной среды). Основное назначение этого понятия в рамках данной работы - описать семантику r-процедур в терминах изменения состояний. Однако, кроме множества динамических переменных, интерпретация которых определяет содержание состояния, нам понадобятся еще два множества: внешние предикатные и функциональные переменные исследуемой модели (PVar) и процедурные переменные (FVar).

Пусть $[States \to States]$ обозначает полную решетку непрерывных функций над пространством состояний States. Val ("означивание нефиксированных параметров") есть пространство таких отображений множества Var в носитель модели M, что $Val(x) \in M_{\tau(x)}$ (т.е. множество означиваний переменных объ-

ектами соответствующих типов). Func обозначает пространство $\mathbf{Val} \to [\mathbf{States} \to \mathbf{States}]$. PVal ("означивание внешних параметров") есть декартово произведение пространств отображений из FVar в Func и из PVal в $\bigcup_{v \in Var} P(M(V))$, т.е. вторая компонента из $e \in \mathbf{PVal}$ соотносит предикатно-функциональным переменным отношения на носителе соответствующих арностей и типов. Таким образом,

$$\mathbf{Val} \subset (\mathbf{Var} \to \mathbf{M}),$$

$$\mathbf{Func} = \mathbf{Val} \to [\mathbf{States} \to \mathbf{States}],$$

$$\mathbf{PVal} = (\mathbf{FVar} \to \mathbf{Func}) \times (\mathbf{PVar} \to \bigcup_{v \in Var} P(M(V))).$$

Императивная (функциональная) семантика r-процедур задается отображением $\mathbf{F}: \mathbf{Proc} \to (\mathbf{PVal} \to \mathbf{Func})$. Содержательный смысл этого отображения таков: выражению из множества Proc при заданных означиваниях внешних (PVal) и нефиксированных (Val из Func) параметров выражения (поведение Val и PVal на всех прочих переменных игнорируется) ставится в соответствие непрерывное отображение на пространстве состояний.

Шесть правил RI0-RI5 определяют семантику выражений $\mathbf{Proc}(\sigma)$ для фиксированной модели $\mathcal{M}$ сигнатуры σ.

Первое из этих правил устанавливает общий принцип отображения $\mathbf{F}[P]\langle e,u\rangle$: $\mathbf{States} \to \mathbf{States}$ ($P \in \mathbf{Proc}$; $u \in \mathbf{Val}$ - означивание переменных из $\mathbf{Unf}(P)$; $e \in \mathbf{PVal}$ - из $\mathbf{Ext}(P)$. Далее I и J обозначают состояния ($I, J \in \mathbf{States}$) такие, что $J = \mathbf{F}[P]\langle e,u\rangle(I)$.

Правило RI0. Если $x \notin \mathbf{Out}(P)$, то $[x]_J = [x]_I \setminus \mathbf{Out}(P)$ и $J[x] = I[x]|[x]_J$.

Другими словами, никакая r-процедура не меняет интерпретации переменных, которые не являются ее выходными. Кроме того, отношение синхронизованности для таких переменных изменяется

минимально: если в I переменная из $Out(P)$ была синхронизована с невыходной переменной, то в состоянии J эти переменные уже не связаны отношением ϑ_J.

2.1. Правила определения синтаксиса и императивной семантики.

Правило RS1. Если $y \in Var$, то $P \equiv empty\ (y) \in Proc$;

$$In(P) = Unf(P) = Ext(P) = \emptyset;\ Out(P) = \{y\};$$

$$Sub(P) = AtSub(P) = \{P\}.$$

Здесь и далее символ "$\equiv$" означает "по определению есть".

Правило RI1. Для $P \equiv empty(x)$ состояние определяется равенством $J[x] = J/x/ = \emptyset$.

Операционно эту процедуру можно трактовать двояко: во-первых, она уничтожает текущее состояние переменной; во-вторых, если x ранее не участвовал в работе, то empty фактически заводит новую переменную с пустым множеством значений.

Правило RS2. Для Σ^+-формулы $\varphi(x_1, \ldots, x_n)$ с множеством свободных переменных $FV(\varphi) = \{x_1, \ldots, x_n\}$ рассмотрим выражение $P \equiv \varphi(m_1, \ldots, m_n)[m_{n+1}, \ldots, m_k]$, где m_i есть порты, заменяющие в синтаксисе Σ^+-формулы все вхождения переменных x_i. Определим $In(P)$, $Out(P)$ и $Unf(P)$ как множества, состоящие из динамических переменных, встречающихся в $m_1, \ldots, m_k$ с префиксами "?", "!" и без префиксов соответственно.

Если эти три множества попарно не пересекаются, то $P \in Proc$ и $Ext(P)$ есть в точности множество предикатно-функциональных переменных формулы φ. Заметим, что допустимы частные случаи, когда $n = 0$ или $n = k$, в последнем случае квадратные скобки можно опустить.

r-процедуры, полученные по правилу RS2, будем называть r-формулами. Термы в r-формулах будем называть r-термами.

Как и в случае RS1 , имеем $Sub(P) = AtSub(P) = \{P\}$.

Правило RI2. r-формула P добавляет новые значения к уже имеющейся интерпретации своих выходных переменных. Строгое определение требует рассмотрения четырех случаев.

Если множество $Out(P)$ пусто, то P может лишь изменить состояние выделенной переменной TruthValue. Если в заданной модели φ истинно при означиваниях e для внешних, u - для нефиксированных и $a \in S_{In(P)}(I)$ - для входных переменных, то P добавляет true к текущему состоянию TruthValue.

Правило RI2.1. Положим

$$F[P]\langle e,u\rangle(I) = I \cup \{TruthValue \to true\},$$

если $\mathfrak{M} \models \exists\, a \in S_{In(P)}(I) P[e,u,a]$,

иначе $F[P]\langle e,u\rangle(I) = I$.

Здесь и далее считается, что для означиваний, собранных в одну пару квадратных скобок выполняются следующие требования:

а) все означивания согласованы между собой (см.§1);

б) если две переменные встречаются в одном порту в процедуре P ,то соответствующие означивания ставят в соответствие этим переменным одни и те же объекты.

Случаи, когда эти требования не выполняются, просто не рассматриваются нами, т.е. практически в правиле RI2.1 квантор существования пробегает только по тем означиваниям из $S_{In(P)}(I)$, которые удовлетворяют указанным правилам "а" и "б".

$S_{In(P)}(I)$ естественно назвать входным потоком (данных) для прроцедуры P в состоянии I.

В случае, если множества $In(P)$ и $Out(P)$ оба пусты, то имеют место:

Правило RI2.2. Положим

$F[P]\langle e,u\rangle(I) = I \cup \{TruthValue \to true\}$,
если $\mathcal{M} \models \varphi[e,u]$, иначе $F[P]\langle e,u\rangle(I) = I$.

Правило RI2.3. Если $W \equiv Out(P) \neq \emptyset$, то определим $\vartheta_J = W \times W \cup (\vartheta_I | (Var\ W))$, т.е. все члены $Out(P)$ становятся синхронизованными в состоянии J и

$$J[W] = S_W(I) \cup$$

$$\cup \{b \in M(W) \mid \mathcal{M} \models \exists a \in S_{In(P)}(I)\ \varphi[e,u,a,b]\}.$$

Аналогично для случая $In(P) = \emptyset$ имеем

Правило RI2.4. Положим

$$J[W] = S_W(I) \cup \{b \in M(W) \mid \mathcal{M} \models \varphi[e,u,b]\}.$$

Для того чтобы объяснить, как работает r-формула, приведем следующее утверждение (при этом ограничимся случаем $Out(P) \neq \emptyset$).

УТВЕРЖДЕНИЕ 8. *Если для* r*-формулы* P *верно* $Out(P) \neq \emptyset$, *то равенство*

$$J[W] = S_{Out(P)}(I) \cup \bigcup_{a \in S_{In(P)}(I)} \{b \in M(W) \mid \mathcal{M} \models \varphi[e,u,a,b]\}$$

равносильно равенству из правила RI2.3.

Исходя из этого, можно сказать, что P вычисляет множество результатов, отвечающее данному a, и добавляет его к уже имеющейся интерпретации переменных из $Out(P)$; далее эта процедура повторяется для каждой подходящей, в указанном выше смысле, n-ки из входного потока данных. Такая трактовка r-формулы позволяет смотреть на нее как на процесс в смысле Кана [10]; предложенная им формализация сетей процессов является основой для развития теории и практики вычислений в стиле "потоков данных".

ЗАМЕЧАНИЕ. Фиктивные порты (в квадратных скобках в синтаксисе r-формул) позволяют синхронизовать переменные из пор-

тов, заменивших логические переменные, с переменными, встречающимися лишь в фиктивных портах. Тем самым фиктивные порты представляют собой механизм управления синхронизованностью переменных.

Правило RS3. Если $R,S \in Proc$ и верно, что

$$(In(R) \cup Out(R)) \cap Unf(S) = \emptyset,$$

$$(In(S) \cup Out(S)) \cap Unf(R) = \emptyset,$$

тогда $P \equiv (R;S) \in Proc$, и $V(P) = V(R) \cup V(S)$, для $V = In, Out, Unf, Ext$:

$$Sub(P) = Sub(R) \cup Sub(S) \cup \{P\};$$

$$AtSub(P) = AtSub(R) \cup AtSub(S).$$

Правило RI3. Пусть $P \equiv (R;S)$, тогда

$$F[P]\langle e,u\rangle(I) = F[S]\langle e,u\rangle(F[R]\langle e,u\rangle(I)).$$

Важно заметить, что связка ";" не обязательно определяет порядок выполнения подпроцедур. Например, $Out(R) \cap In(S) \neq \emptyset$ указывает только на то, что результаты работы R будут поступать на вход процедуры S. Например, как мы уже указывали, исполнение r-формулы есть итеративная обработка множества входных данных с итеративным вычислением результатов, поэтому требовать, чтобы S начинала работу после окончания работы r-процедуры R, неразумно. Если же $Out(R) \cap In(S) = \emptyset$, то исполнение подпроцедур можно производить независимо, так как R не может влиять на ход исполнения S.

Заметим, что наша семантика не включает в себя традиционного для денотационных семантик языков программирования понятия продолжения (continuation). Это связано с нашим предположением о том, что любая процедура может успешно работать в любом состоянии.

Правило RS4. Если $R \in Proc$, $z \in Unf(R)$, $l \in Var \setminus Unf(R)$ и $\tau(I) = list$, тогда $P \equiv {}^{*}z \leftarrow$

$\leftarrow I.R \in Proc$ и

$$Ext(P) = Ext(R);\ Unf(P) = Unf(R)\setminus\{z\};$$

$$Out(P) = Out(R);\ In(P) = In(R)\cup\{1\};$$

$$Sub(P) = Sub(R)\cup\{P\};\ AtSub(P) = AtSub(R).$$

Эта процедура определяет второй вариант итерации в нашем языке. Первый вариант был представлен r-формулами, которые трактовались как "применение (одной) процедуры определения истинности к потоку входных данных". Второй вариант следует трактовать как "применение нескольких процедур к одному потоку данных". Как организовать такую итерацию? Простым и естественным видится следующий способ: использовать некоторую нефиксированную переменную z r-процедуры R как индексную. Тогда последовательность процедур можно задать списком элементов $M_{\tau(z)}$, которые будут означивать z в экземплярах R. Эти экземпляры R должны применяться к данным последовательно, согласно порядку элементов в списке, что и отражено в семантике.

Правило RI4. Семантику r-процедуры $P \equiv {}^{*}z \leftarrow I.R \in Proc$, $z \in Unf(R)$, $1 \in Var\setminus Unf(R)$ и $\tau(1) = list$, легче всего задать так, как это было сделано в утверждении 8:

$$F[P]\langle e,u\rangle(I) = \bigcup_{a \in I[1]} F^{z,1,a}[R].$$

Здесь $F^{z,1,a}[R]$ обозначает

$$F[R]\langle e,u\setminus z \leftarrow b_n\rangle(F[R]\langle e,u\setminus z \leftarrow b_{n-1}\rangle(\ldots$$

$$\ldots F[R]\langle e,u\setminus z \leftarrow b_1\rangle(I\setminus a)\ldots)),$$

если список $a(1) = \langle b_1,\ldots,b_n\rangle$ и просто I, если $a(1)$ - пустой список. Запись $u\setminus z \leftarrow b$ обозначает отображение типа $Var \to M$, совпадающее с u на всех переменных, кроме z, и ставящее в соответствие переменной z элемент b.

Правило RS5. Если $f \in FVar$, то $f \in Proc$;

$$Ext(f) = \{f\};\quad In(f) = Out(f) = Unf(f) = \emptyset.$$

Таким образом, мы резервируем в языке r-процедур возможность параметризации процедур другими процедурами. Семантика в этом случае определяется следующим образом.

Правило RI5. Положим $F[f]\langle e\rangle = e(f)$.

Полезно заметить следующие тривиальные следствия из правил синтаксиса.

УТВЕРЖДЕНИЕ 9. *Всякая* Σ^+*-формула есть* r*-формула и, следовательно,* r*-процедура.*

УТВЕРЖДЕНИЕ 10. *Для всякой процедуры* $R \in Proc$ *множество* $Ext(R)$ *не пересекается с множествами* $In(R)$, $Out(R)$ *и* $Unf(R)$.

На этом определение синтаксиса и императивной (F) семантики языка заканчивается. Для того, чтобы определение семантики было корректным, необходимо доказать .

УТВЕРЖДЕНИЕ 11. *Для произвольных* $P \in Proc$, $e \in PVal$ *и* $u \in Val$ *оператор* $F[P]\langle e,u\rangle : States \to States$ *непрерывен.*

2.2. Декларативная семантика. С теоретико-модельной точки зрения n-арный предикат $\phi(x_1, \ldots, x_n)$ есть n-арное отношение, т.е. подмножество $M(\{x_1, \ldots, x_n\})$.Здесь нам хотелось бы предложить такую семантику для r-процедур, которая поставила бы в соответствие процедуре некоторое отношение, и тем самым сделала понятия Σ^+-формул и r-процедур близкими. (Подобная теоретико-множественная семантика была описана для другого расширения класса Σ^+-формул в [7].) Чтобы две семантики (описанная выше императивная (F) и предлагаемая декларативная (S)) коррелировали между собой, определение второй семантики дается через первую. Таким образом, процедуры получат декларативный смысл посредством S аналогично тому, как семантика F придала императивный смысл Σ^+-формулам.

ОПРЕДЕЛЕНИЕ 5. S есть оператор $Proc \to (PVal \to \to \bigcup_{W \subset Var} M(W))$, заданный следующим правилом (0 - пустое состояние):

Правило R6. Если $Unf(P) = \emptyset$, то

$S[P]\langle e\rangle = true$ при $F[P]\langle e\rangle(0)/TruthValue/ = \{true\}$,

$S[P]\langle e\rangle = false$ при $F[P]\langle e\rangle(0)/TruthValue/ = \emptyset$.

Если $Unf(P) \neq \emptyset$, то

$$S[P]\langle e\rangle = \{ u \mid \neg F[P]\langle e,u\rangle(0) \approx 0\}.$$

Таким образом, изменение состояния оператором $F[P]\langle e,u\rangle$ трактуется как "истинность" процедуры P на n-ке u в заданной модели.

Несколько простых фактов хорошо иллюстрируют корреляцию двух семантик.

УТВЕРЖДЕНИЕ 12. *Пусть семантика* F r*-процедур* R *и* P *совпадает* (т.е. совпадают отображения $F[P]$ и $F[R]$), *тогда совпадает и семантика* S *этих процедур.*

УТВЕРЖДЕНИЕ 13. *Теоретико-множественная семантика* Σ^+*-формулы* φ *при заданном означивании предикатных переменных* e *совпадает с* $S[\varphi]\langle e\rangle$.

Таким образом, на семантику S можно смотреть как на распространение теоретико-множественной семантики языка Σ^+-формул на язык r-процедур.

УТВЕРЖДЕНИЕ 14. *Пусть* P - r*-формула,* $W = Unf(P)$ *и* $V = Unf(P) \setminus W$, *тогда верно*

$$\forall e \in PVal \quad \forall u \in M(V) \quad \{c \in M(W) \mid (c,u) \in S[P]\langle e\rangle\} = S_W(F[!W.P]\langle e,u\rangle(0)),$$

где $!W.P$ *обозначает процедуру, полученную из* P *синтаксической заменой всех портов с вхождениями переменных* $x \in W$ *на порты* $!x$.

В частности, имеет место

СЛЕДСТВИЕ. *Для* r*-формулы* P *верно*

$$\forall e \in PVal\ \forall u \in M(V)\ S[P]\langle e\rangle = S_W(F[!Unf(P).P]\langle e,u\rangle(0)).$$

Другими словами, декларативная семантика r-процедуры может быть установлена через ее императивную семантику, которая, в свою очередь, является прообразом операционной семантики реализации языка r-программ. При наличии такой реализации мы сможем вычислять денотационную семантику Σ^+-формул и, следовательно, Σ^+-программ.

2.3. Примеры. Ниже будут приведены несколько простых r-процедур, предназначенных для иллюстрации синтаксиса, семантики и отчасти выразительных возможностей описанного языка. В приведенных примерах not, or и and обозначают логические связки, exist - неограниченный квантор существования, exist... ...in и for_all...in - ограниченные кванторы существования и всеобщности; считается допустимым "длинное" равенство ... = = ... = ... Кроме того, используются удобные сокращенные записи: $(P_1;P_2;\ldots;P_n)$ означает $(\ldots(P_1;P_2;\ldots;P_n)$, а $empty(y_1,\ldots,y_n)$ означает $(empty(y_1);\ldots;empty(y_n))$.

1. Просто записываются операции для объединения и пересечения множеств данных, хранящихся в нескольких переменных, в одно множество: (TRUE[?A!C]; TRUE[?B!C]) - к интерпретации переменной C добавляются интерпретации переменных A и B; TRUE[?A!C, ?B!C] - к интерпретации переменной C добавляется пересечение множеств, интерпретирующих переменные A и B.

Заметим, что первой конструкции эквивалентны r-процедуры (?A = !C; ?B = !C) (?A = !C or ?B = !C) и т.д., а второй - ?A = ?B!C, !C = ?A = ?B (?A = !C and ?B= = !C).

2. r-процедура (empty(C); TRUE[?A!C]) есть фактически оператор присваивания "C:= A".

3. Если P и R - r-формулы, причем Out(P)= ∅, то (empty(TruthValue); P; R[?TruthValue]) фактически есть условный оператор "if P then R".

4. Если less - отношение порядка, то формула

(X = Y and less(Y,Z)) or (X = Z and less(Z,Y))

означает, что X есть минимум из значений Y и Z .Наше знание о том, что с императивной точки зрения при данных ("входных") Y и Z лучше сначала проверять отношение неравенства, можно записать так

(empty(TruthValue);less(?Y,?Z);?Y=!X[?TruthValue];
empty(TruthValue);less(?Z,?Y);?Z=!X[?TruthValue]).

5. Формула, означающая, что X есть наибольший общий делитель Y и Z , может иметь такой вид:

mod(X,Y) and mod(X,Z) and
for_all W in interval(s(X),mod(Y,Z))
(not(mod(W,Y) and mod(W,Z))).

Здесь mod - отношение "делит"; s - отношение "следующий"; interval дает список, состоящий из всех целых чисел из указанного отрезка; min - "минимум".

Как этой r-формуле можно придать более императивный вид, зная, например, что Y и Z даны? Используем конструктор "*":

(empty(X1,X2,X3,Y1,Y2,Y3,Z2,Z3); (1)
mod(!X1,?Y!Y1); (2)
mod(?X1 !X2,?Z!Z2)[?Y1!Y2]; (3)
'L = interval(s(?X2!X3),min(?Y2!Y3,?Z2!Z3)); (4)

```
*W ← L.(empty(X4);                                    (5)
        not mod(W,?Y3)[?X3!X4];                       (6)
        not mod(W,?Z3)[?X3!X4];                       (7)
        empty(X3); TRUE[?X4!X3]);                     (8)
TRUE[?X3!X]).                                         (9)
```

Рассмотрим подробнее, как построена и работает эта r-процедура. Предположим, что в исходном состоянии переменным Y и Z приписаны пары натуральных чисел. Покажем, что в результате работы процедуры переменная X принимает множество наибольших общих делителей этих пар, причем результат достигается определенной алгоритмизацией приведенного декларативного определения наибольшего общего делителя. Как видно, процедура предлагает сначала найти все величины, удовлетворяющие первой части формулы, т.е. все общие делители значений Y и Z . В строке (2) X1 принимает все делители каждого значения Y , причем, чтобы можно было определить, каким значениям X1 соответствуют какие значения Y , вся интерпретация Y передается в Y1 . Тем самым после выполнения строки (2) состояние переменных X1 и Y1 таково, что они содержат всевозможные пары чисел таких, что первое есть делитель второго и среди вторых есть все значения из исходной интерпретации Y . Аналогична работа подпроцедуры в строке (3). После ее выполнения состояния X2,Y2 и Z2 таковы, что они содержат тройки, где первое число есть общий делитель второго и третьего. Используя тот же прием "фильтра" (порт вида ?x!y), строка (4) добавляет к этим тройкам четвертую компоненту - список, ограничивающий квантор формулы. Подпроцедуры в строках (5)-(8) переписывают в более императивный вид квантор всеобщности, вычеркивая те значения X3 , которые оказываются не наибольшими делителями.

Важно отметить, что описанную процедуру можно получить из исходного определения в виде формулы автоматически. Таким образом, наибольший общий делитель может находиться без знания какого-либо алгоритма его вычисления, но исходя только из его определения.

З а к л ю ч е н и е

Введение формализма r-процедур в данной работе преследовало весьма конкретные цели. С точки зрения построения операционной семантики Σ^+-программ язык r-процедур может служить промежуточным звеном при переходе от декларативного к императивному описанию задачи, а также использоваться при анализе различных аспектов процессов нахождения ее решения.

В перспективе мы надеемся подробно изложить варианты схем многошаговой трансляции Σ^+-программ с использованием r-процедур, пока же мы ограничимся их кратким описанием. В первую очередь надо заметить, что представленный язык позволяет лишь внести императивные аспекты в декларативный формализм. Однако служить в качестве императивного языка, который "ближе" к реализации,чем Σ^+-программы, он не может, так как класс r-процедур содержит в себе класс Σ^+-формул.

Тем не менее, пользуясь различными синтаксическими критериями, в классе r-процедур можно выделять подклассы процедур, в которых доминирует тот или иной способ описаний. С практической точки зрения наиболее интересным является класс fr-процедур ("плоские r-процедуры", [4]). Он состоит из процедур,имеющих минимально возможную декларативную часть и поэтому носящих отчетливо императивный характер. Следовательно, fr-процедуры наиболее близки (из всех r-процедур) к конструкциям реальных языков программирования. Поэтому мы предполагаем использовать fr-процедуры как средство перехода от r-процедур (и тем самым Σ^+-формул) к хорошо изученным и реализованным

языкам. В работе [4] показывается существование алгоритма,осуществляющего переписывание произвольной r-процедуры в fr-процедуру с эквивалентной семантикой S .

Итак, первым шагом трансляции предлагается переписывание Σ^+-схем в семантически эквивалентные вычислительные схемы,синтаксис которых основан на fr-процедурах. Существенным фактом является неединственность представления (нетривиальной) декларативной программы посредством вычислительных схем. В связи с этим возникает понятие стратегии трансляции, охватывающее (достаточно регулярные) способы трансформации программ.

Несмотря на императивный характер описания задачи, вычислительные схемы еще не являются средством реализации семантического программирования. Конечно, возможна разработка интерпретатора, исполняющего эти программы, однако только сравнительно узкий класс задач может быть успешно решен таким способом. Более реальным видится другое направление работы: продолжение трансляции - перевод fr-языка в реальные средства программирования. И с практической, и с теоретической точек зрения мы считаем полезным трансляцию в "динамические сети (недетерминированных) процессов", формализм, основанный на идеях,изложенных в [11].Эта работа запланирована на ближайшее будущее.

Автор считает необходимым выразить свою благодарность Д.И.Свириденко за большую помощь в подготовке данной работы.

Литература

1. ГОНЧАРОВ С.С., СВИРИДЕНКО Д.И. Σ-программирование // Логико-математические проблемы МОЗ.-Новосибирск.-1985.-Вып.107: Вычислительные системы.-С.3-29.

2. Их же. Математические основы семантического программирования// Докл.АН СССР.-1986.-Т.289,№ 6.-С.1324-1328.

3. Их же. Σ-программы и их семантика // Логические методы в программировании.-Новосибирск.-1987.-Вып.120: Вычислительные системы.-С.24-51.

4. КОТОВ С.В. К операционной интерпретации Σ^+-программ. r-программы и их семантика// Настоящий сб.- С. 160-184.

5. Логическое программирование/ Отв.ред.В.Н.Агафонов.-М.: Мир, 1988.- 366 с.

6. СВИРИДЕНКО Д.И. Теория семантического программирования / Автореф.дис... д-ра физ.-мат.наук.- Новосибирск,1989.-17 с.

7. Его же.Об одном обогащении языка Σ^+-программ// Логические методы в программировании.-Новосибирск.-1987.-Вып.120: Вычислительные системы.- С.97-104.

8. ХОГГЕР К.Д. Введение в логическое программирование. - М.: Мир, 1988.- 348 с.

9. APT K.R., van EMDEN M.H. Contributions to the Theory of Logic Programming// J.of the ACM.-1982.-Vol.29.-P.841-863.

10. KAHN G. The Semantics of a Simple Language for Paral - lel Programming// Proceedings of IFIP'74.- Amsterdam: North-Holland, 1974.-P.471-475.

11. KAHN G., MacQUEEN D.B. Coroutines and Networks of Parallel Processes=// Proceedings of IFIP'77.-Amsterdam:North-Holland,1977.-P.993-998.

Поступила в ред.-изд.отд.

15 мая 1990 года