

Шилов Николай Вячеславович

**ОСНОВЫ СИНТАКСИСА, СЕМАНТИКИ,
ТРАНСЛЯЦИИ И ВЕРИФИКАЦИИ ПРОГРАММ**

Содержание

Содержание.....	3
Часть I. ВВЕДЕНИЕ.....	5
Лекция 1. Введение в проблематику языков программирования....	5
Лекция 2. Неформальное введение в формальную верификацию.....	20
Лекция 3. Воспоминания о теории множеств и математической логике – I.....	34
Лекция 4. Воспоминания о теории множеств и математической логике - II.....	50
Лекция 5. Основы автоматического доказательства теорем.....	66
Лекция 6. Методы верификации пропозициональных формул....	76
Лабораторная работа 1. Алгоритмы верификации Формул пропозициональной логики.....	88
Часть II. СИНТАКСИС.....	93
Лекция 7. Синтаксис языков программирования.....	93
Лекция 8. Контекстно-свободные языки.....	109
Лекция 9. Основы лексического и синтаксического анализа (на примере языка HeMo).....	121
Лекция 10. Корректность и сложность синтаксического анализа (на примере языка HeMo).....	140
Лабораторная работа 2. Грамматический разбор и внутреннее представление HeMo-программ.....	150
Часть III. СЕМАНТИКА И ТРАНСЛЯЦИЯ.....	154
Лекция 11. Семантика типов данных (на примере языка HeMo).....	154
Лекция 12. Виртуальная HeMo машина.....	170
Лабораторная работа 3. Интерпретация языка виртуальной HeMo-машины.....	181
Лекция 13. Введение в семантику программ (на примере языка HeMo).....	185
Лекция 14. Введение в компиляцию языков программирования (на примере языка HeMo).....	205
Лабораторная работа 4. Трансляция вычислительных HeMo-программ в вычислительные программы виртуальной HeMo-машины.....	218
Лекция 15. «Универсальный» язык программирования.....	219
Часть IV. На пути к верификации.....	236

Лекция 16. Определение аксиоматической семантики вычислительных программ (на примере языка HeMo).....	236
Лабораторная работа 5. Грамматический разбор и внутреннее представление формальных троек Хоара для языка HeMo.....	258
Лекция 17. Семантика аксиоматической семантики вычислительных программ (на примере языка HeMo).....	261
Лекция 18. От надежности – к полноте аксиоматической семантики (на примере языка HeMo).....	278
Лекция 19. Полнота аксиоматической семантики. Идеализированный язык Pascal и его аксиоматическая семантика.....	295
Лекция 20. Основы верификации вычислительных программ (на примере языка HeMo и I-Pascal).....	314

Часть I. ВВЕДЕНИЕ

Лекция 1. Введение в проблематику языков программирования

«Великая проблема программирования» Антони Хоара

Самая престижная международная научная премия в области программирования – это премия имени Алана М. Тьюринга. Она учреждена Ассоциацией Машинных Вычислений¹ в честь одного из основоположников теории алгоритмов и машинных вычислений – британского учёного Алана М. Тьюринга (1912–1954). В 1980 г. этой премии был удостоен другой британский учёный, наш современник, Антони Хоар² за «основополагающий вклад в определение и разработку языков программирования».

По-видимому, признание заслуг А. Хоара среди специалистов может служить весомым аргументом даже для новичков в программировании для того, чтобы прислушаться к тому, что Антони Хоар называет Великими проблемами программирования как науки³:

¹ Association for Computing Machinery (ACM, см.: <http://www.acm.org/>). О самой премии и полный список лауреатов см. <http://www.acm.org/awards/taward.html>. Лекции лауреатов этой премии за первые 20 лет её существования, прочитанные во время церемонии вручения, были опубликованы на русском языке (см.: Лекции лауреатов премии Тьюринга: Пер. с англ. М.: Мир, 1993).

² Антони Хоар обучался в Московском государственном университете в 1958–1959 гг.

³ Hoare C.A.R. The Verifying Compiler: A Grand Challenge for Computing Research. Proceedings of A. P. Ershov Memorial Conference «Perspectives of System informatics», Novosibirsk, Russia, 2003.// Lecture Notes in Computer Science. Springer Verlag. 2003. Vol.2890. P.1-12. На URL http://research.microsoft.com/~thoare/The_Verifying_Compiler.ppt можно найти обновлённые слайды этой лекции.

Проблема		Статус
Название	Вариант формулировки	
P vs. NP	Существует ли нет полиномиальный по сложности алгоритм проверки выполнимости булевских формул	Открытая проблема ⁴
Тест Тьюринга	Тест, предложенный А. Тьюрингом в 1950 г. в статье «Вычислительные машины и разум» для проверки, является ли разумным в человеческом смысле слова «искусственный интеллект», реализованный программно на вычислительной машине	Может быть невыполнимым
Шахматный тест	Шахматная компьютерная программа, выигрывающая у чемпиона мира по шахматам	Решена (1997)
Верифицирующий транслятор	Компьютерная программа, которая переводит (транслирует) написанные человеком программы с языка высокого уровня в эквивалентные программы, пригодные для непосредственного исполнения на вычислительной машине, и при этом доказывает специфицированные человеком математические утверждения о свойствах транслируемых программ	В процессе решения

По мнению А. Хоара, для того, чтобы какая-либо научная проблема могла называться Великой, она должна удовлетворять следующему ряду требований:

- проблема должна носить фундаментальный характер, её решение будет иметь прежде всего научное значение;
- решение может иметь и прикладное значение, но в масштабах всего человечества;
- решение проблемы может потребовать десятилетий скоординированных усилий учёных и коллективов мирового класса;
- есть чёткие критерии успеха или провала в решении проблемы.

⁴ В 2000 г. Учёный Совет Clay Mathematics Institute (CMI, см.: <http://www.claymath.org/>) определил список из 7 нерешённых математических проблем для математики XXI века и учредил премию по 1 млн. долларов за решение каждой из названных проблем (см.: <http://www.claymath.org/millennium/>). В этом списке значится и проблема P vs. NP (см.: http://www.claymath.org/millennium/P_vs_NP/).

А. Хоара приводит примеры решённых Великих научных проблем: полёт человека на Луну, расшифровка генома человека, доказательство Великой теоремы Ферма. А в качестве примера «проваленной» Великой научной проблемы А. Хоар приводит проблему лечения рака: как оказалось, различных причин развития рака слишком много, а механизмы его развития слишком сложны для того, что бы надеяться на панацею.

Завершим эту водную часть курса коротким комментарием, почему успешное решение проблемы Верифицирующего Транслятора может иметь огромное значение в эпоху информационных технологий. Дело в том, что проверка синтаксической правильности программы и её трансляция в исполняемый машинный код осуществляется полностью автоматически. Но этап отладки программ до сих пор плохо автоматизирован. Воспользуемся следующим образным сравнением В.А. Захарова⁵: «Средства отладки программы, которые входят в состав всякой развитой системы программирования, могут помочь разработчику в той же мере, в какой лопата оказывает помощь кладовищу. Главные вопросы - где копать и стоит ли копать вообще? Здесь скорее пригодились бы устройство наподобие металлоискателя. Но хороший металлоискатель - это тонкий инструмент, требующий знания его устройства и умелого обращения. Так что кроме лопаты неплохо бы использовать и автоматические системы поиска ошибок и проверки правильности функционирования программ...»

Об особенностях данного курса лекций

Курс «Основы синтаксиса, семантики, трансляции и верификации программ» построен как введение в последнюю из перечисленных выше Великих научных проблем программирования – проблему Верифицирующего Транслятора. В то же время курс соответствует российским вузовским стандартам по основам трансляции языков программирования по специальностям программистского и математического профиля, а также информационных технологий. С точки зрения мировых образовательных стандартов по программированию курс соответствует ACM Computing Curricula по направлению Computer Science: Computer science body of knowledge with core topics – Programming languages. (Имеется русский перевод⁶.)

Совместное и взаимосвязанное изложение основ трансляции языков программирования и основ формальной верификации программ является

⁵ Из предисловия переводчиков к книге: Кларк Э. М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: Издательство Московского центра непрерывного математического образования, 2002.

⁶ Рекомендации по преподаванию информатики в университетах. СПб-б.: Издательство С.-Петербургского университета, 2002 (под ред. В.Л. Павлова и А.А. Терехова).

главной особенностью предлагаемого курса. Помимо этого, особое внимание уделяется семантике языков программирования. К сожалению, имеющаяся на русском языке учебная литература, посвящённая трансляции языков программирования, не уделяет внимания современной семантике языков программирования и семантическим вопросам трансляции⁷.

Другой особенностью курса является построение его вокруг модельного языка недетерминированного программирования НеМо. Такой подход позволяет сконцентрироваться и рассказать доступно и точно о главном в трансляции, семантике и верификации программ, не отвлекаясь на особенности того или иного конкретного реального языка программирования, той или иной реальной вычислительной машины⁸. Заметим, что опыт построения курса по верификации программ на некоем модельном языке за рубежом имеется. (Можно сослаться, например, на книгу профессора Кембриджского Университета Майкла Гордона⁹.)

Курс снабжен пятью контрольными лабораторными работами. Каждая контрольная лабораторная работа посвящена решению определённой задачи, которая является составной частью верифицирующего транслятора, как то, например: лексический и синтаксический анализатор программ и их спецификаций, интерпретатор виртуальной машины и верификатор пропозициональных формул. Каждая лабораторная работа включает в себя разработку (дизайн) и программную реализацию некоторого алгоритма. Лабораторные работы расположены в соответствии с порядком изложения материала в курсе по темам (через 2–5 лекции) и не требуют никаких дополнительных знаний (кроме навыков проектирования и реализации алгоритмов), и поэтому в курсе нет отдельных задач и упражнений после каждой лекции. Главная учебно-методическая цель контрольных лабораторных работ – дать возможность учащимся на практике проверить и применить полученные знания, а преподавателю – проверить уровень усвоения материала курса.

НеМо: неформальное введение

НеМо – Недетерминированный Модельный язык императивного программирования. Каждая программа на языке НеМо состоит из описаний и тела. Каждое описание имеет вид

⁷ См., например: Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: в 2 т. М.: Мир, 1978; Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. М.: Издат. дом Вильямс, 2001.

⁸ Подобный подход был принят Д. Кнудом в его знаменитой книге «Искусство программирования для ЭВМ» при описании эффективных алгоритмов для компьютера: он просто предложил модельный компьютер MIX.

⁹ Gordon M.: Programming language Theory and its Implementation. Prentice Hall, 1988.

VAR ПЕРЕМЕННАЯ : ТИП,

где допустимые типы – это предопределённый целый тип INT или массив, сконструированный из ранее других типов

(ТИП_1 ARRAY OF ТИП_2).

Тело любой программы состоит из операторов двух типов: присваиваний и тестов (проверок). Вычислительная программа – это программа, которая за конечное время преобразует начальные (входные) данные в заключительные (выходные) значения, а резидентная программа – это программа, которая должна работать бесконечно долго и может в процессе многократно получать входные значения и многократно выдавать выходные значения¹⁰.

Тип INT – это математические целые числа \mathbf{Z} от $-\infty$ до $+\infty$ с обычными операциями сложения и вычитания, умножения и деления¹¹. Тип (ТИП_1 ARRAY OF ТИП_2) состоит из «таблиц», ячейки которых «пронумерованы» значениями типа ТИП_1, а в этих ячейках могут храниться неопределённые значения или значения, которые имеют ТИП_2. Иными словами, (ТИП_1 ARRAY OF ТИП_2) состоит из частичных функций¹², представленных «таблично», область определения которых это ТИП_1, а область изменения – ТИП_2. Так, например, тип (INT ARRAY OF INT) состоит из целочисленных функций целого аргумента. Легко видеть, что ((INT ARRAY OF INT) ARRAY OF INT) не совпадает с (INT ARRAY OF (INT ARRAY OF INT)): в первом случае значениями являются «функционалы», которые сопоставляют каждой функции целое число, а во втором – «двумерные массивы», которые можно представлять как последовательности «массив одномерных массивов».

На типе (ТИП_1 ARRAY OF ТИП_2) определены две операции: вычисление значения функции в точке (т. е. компоненты массива по индексу), и изменение значения функции в точке. Приведём примеры этих операций. Если Y – переменная типа (INT ARRAY OF INT), то APP(Y,56) – это пример применения операции вычисления функции Y в точке 56, которая возвращает это значение в качестве своего результата. (Sic! В HeMo можно использовать обозначение Y[56] вместо APP(Y, 56) как «синтаксический сахар».) При тех же условиях UPD(Y, 56, 100) – это пример применения опе-

¹⁰ Резидентные программы иногда называют «реактивными», так как смысл их непрерывной работы – «реакция», обработка запросов пользователя или устройств компьютера. Пример резидентной программы – операционная система.

¹¹ Сейчас мы не уточняем, что будет результатом при делении на ноль.

¹² Частичная функция – это не всюду определённая функция. Например функция целого аргумента $\text{tg}(\pi \times k/2): \mathbf{Z} \rightarrow \mathbf{Z}$ определена для всех чётных чисел и неопределена для всех нечётных чисел.

рации изменения значения функции Y в точке 56 на новое значение 100; эта операция возвращает функцию, которая всюду совпадает с Y кроме точки 56, в которой эта новая функция принимает значение 100. (Sic! $UPD(Y, 56, 100)$ – операция, возвращающая «безымянную функцию», но не оператор присваивания $Y[56] := 100$, изменяющий саму функцию Y .)

Всякое присваивание имеет вид

ПЕРЕМЕННАЯ := ВЫРАЖЕНИЕ,

причём ПЕРЕМЕННАЯ и ВЫРАЖЕНИЕ должны иметь один и тот же тип (в соответствии с описаниями). Смысл оператора присваивания – изменение значения переменной. Например, $X := 28$ – это обычный оператор присваивания переменной X типа INT нового значения 28. А присваивание $Y := UPD(Y, 56, 100)$ – это то же, что в традиционной нотации записывается как $Y[56] := 100$. (Sic! В HeMo можно использовать оператор присваивания элементу массива как синтаксический сахар: например, $Y[56] := 100$ вместо $Y := UPD(Y, 56, 100)$.)

Всякий тест имеет вид

(УСЛОВИЕ ?),

где УСЛОВИЕ – это пропозициональная комбинация¹³ целочисленных (не)равенств, например: $X + 29 = 34 \text{ AND } Y[56] > 24$. Смысл теста состоит в том, что если при проверке УСЛОВИЯ оно оказывается истинно (верно), то ничего не происходит; в противном случае тест создает исключительную ситуацию аварийной остановки, аннулирует произведенные вычисление, которое привело к проверке теста. Значит, в случае, когда УСЛОВИЕ верно, тест (NOT УСЛОВИЕ)? аннулирует всякое вычисление, в котором участвует, а тест (УСЛОВИЕ ?) никак не влияет на любое вычисление в котором он проверяется. Примеры использования тестов приведены ниже (после определения правил построения тел программ на языке HeMo).

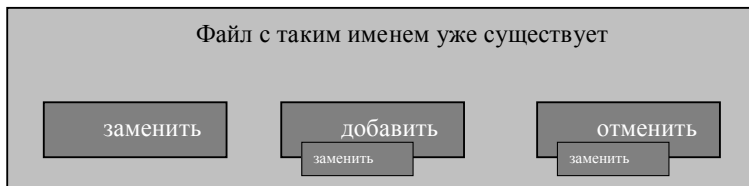
Большие программы строятся из более простых при помощи конструкторов последовательного исполнения «;», недетерминированного выбора « \cup » и недетерминированной итерации «*», а именно: если α и β – два тела программ, то $(\alpha ; \beta)$, $(\alpha \cup \beta)$ и (α^*) – три тела программ, которые соответствуют:

- последовательному исполнению сначала α , потом β ,
- выбору одного из возможных исполнений α или β ;
- исполнению α последовательно 0 раз, или 1 раз, или 2 раза и т. д..

По-видимому, понятие последовательного исполнения не нуждается в неформальных пояснениях. А вот недетерминированный выбор и недетерминированная итерация нуждаются в комментариях.

¹³ при помощи конъюнкций, дизъюнкций и отрицания

Представим себе ситуацию, когда надо «запрограммировать» реакцию человека на появление следующего выпадающего меню при сохранении файла:

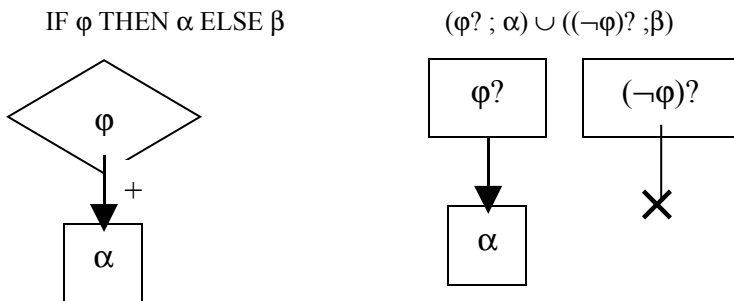


В таком случае нельзя приписать никакой определённой вероятности выбора ни одной из клавиш, а правильнее будет использовать недетерминированную программу ($\text{ВЫБОР} := \text{заменить} \cup \text{ВЫБОР} := \text{добавить} \cup \text{ВЫБОР} := \text{отменить}$).

Представим себе другую ситуацию, когда надо вычислить произвольное целое число, превосходящее начальное значение переменной X , причём любое число равновозможно. В данном случае вероятностный подход неприменим, поскольку «равновозможно» не имеет количественной меры. В качестве правильного варианта для равновозможного вычисления любого целого числа предлагается недетерминированная программа $((X := X + 1)^*)$, которая генерирует новые числа, но может остановиться (завершить вычисления) в любой момент.

С использованием тестов можно познакомиться на примере тела программы вида $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$. Несмотря на наличие недетерминированного выбора в $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ между $(\phi? ; \alpha)$ и $((\neg\phi)? ; \beta)$, если ϕ – верно, то выполняется только α , а вычисление β не допускается, так как ему предшествует проверка $(\neg\phi)?$. Аналогично, если ϕ – ложно, то в $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ выполняется только β , а всякое вычисление α не допускается, поскольку ему предшествует проверка $\phi?$. Поэтому в вычислительных программах, в которых интересен только вход-выход, конструкция $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ эквивалентна обычной конструкции $(\text{IF } \phi \text{ THEN } \alpha \text{ ELSE } \beta)$ и в таких программах её можно использовать как «синтаксический сахар».

Иначе обстоит дело с резидентными программами, в которых значение имеет процесс исполнения программы. Например, в случае, когда ϕ верно, у стандартной конструкции $(\text{IF } \phi \text{ THEN } \alpha \text{ ELSE } \beta)$ есть только один сценарий исполнения, а у конструкции $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ – два:



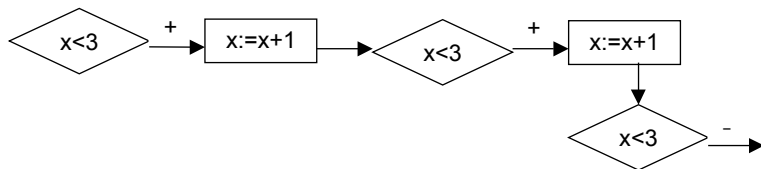
Аналогично обстоит дело в случае, когда верно $\neg\phi$. Поэтому для резидентных программ конструкции (IF ϕ THEN α ELSE β) и $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ нельзя считать эквивалентными.

Другой пример использования тестов – тело программы $((\phi? ; \alpha)^* ; (\neg\phi)?)$. Это тело начинается с исполнения недетерминированной итерации $(\phi? ; \alpha)^*$, которая (в принципе) допускает исполнение тела $(\phi? ; \alpha)$ любое количество раз. Однако каждое повторение начинается с проверки, которая аннулирует всякие вычисления как только ϕ ложно. Поэтому $(\phi? ; \alpha)^*$ допускает исполнение тела $(\phi? ; \alpha)$ любое количество раз, пока ϕ истинно. Однако $((\phi? ; \alpha)^* ; (\neg\phi)?)$ заканчивается проверкой $(\neg\phi)?$, которая аннулирует всякое исполнение, когда верно ϕ . Поэтому вся конструкция $((\phi? ; \alpha)^* ; (\neg\phi)?)$ повторяет α пока верно ϕ и прекращает итерации, как только ϕ становится ложным. Таким образом, в вычислительных программах, в которых интересен только вход-выход, конструкция $((\phi? ; \alpha)^* ; (\neg\phi)?)$ эквивалентна обычной конструкции (WHILE ϕ DO α) и в таких программах её можно использовать как «синтаксический сахар».

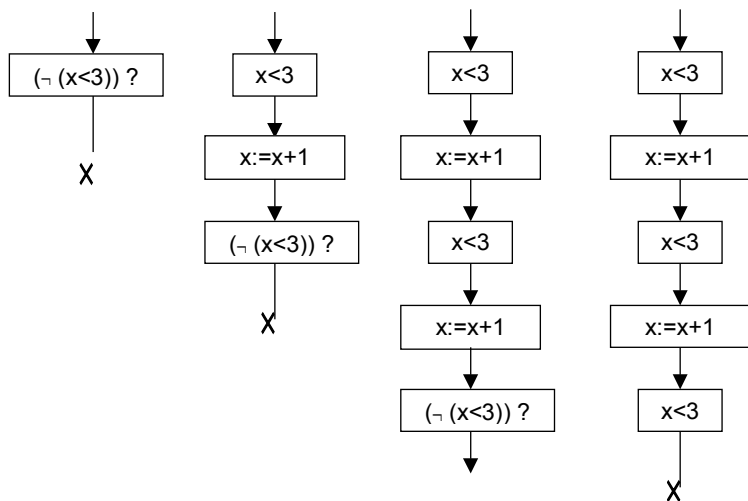
Например, $(x < 3? ; x := x + 1)^*$ при начальном значении $x_{\text{нач}} = 1$ может быть исполнено 0, 1 и 2 раза и остановиться с заключительным значением $X = 1, 2, 3$ соответственно. Проверка $(\neg(x < 3))?$ аннулирует всякое вычисление, в котором значение x меньше 3. Следовательно, $((x < 3? ; x := x + 1)^* ; (\neg(x < 3))?)$ итерирует $(x := x + 1)$ ровно 2 раза и заканчивает работу. С точки зрения отношения вход-выход эта конструкция действительно эквивалентна стандартному циклу WHILE $(x < 3)$ DO $x := x + 1$.

Однако для резидентных программ конструкции $((\phi? ; \alpha)^* ; (\neg\phi)?)$ и (WHILE ϕ DO α) не являются эквивалентными. Сравним, например, процессы, которые порождаются стандартной конструкцией WHILE $(x < 3)$ DO $x := x + 1$ и конструкцией $((x < 3? ; x := x + 1)^* ; (\neg(x < 3))?)$ языка HeMo при начальном значении $x = 1$:

WHILE ($x < 3$) DO $x := x + 1$



$((x < 3? ; x := x + 1)^* ; (\neg(x < 3)))?$



Логические спецификации: неформальное введение

Программы описывают преобразования данных, но не смысл этих преобразований. Смысл преобразований (т. е. семантику программ) описывают спецификации программ. Поэтому наряду с языками программирования существуют языки спецификаций программ. Так же как существуют разные парадигмы языков программирования (например, императивная, функциональная и логическая), существуют разные парадигмы спецификации программ (логическая, алгебраическая и т. д.). Познакомимся с логическими спецификациями на примере детерминированных программ, у которых

каждый шаг исполнения однозначно определяется текущей конфигурацией (т. е. значениями переменных и исполняемым оператором).

Определение 1. Вычислительная программа – это программа, которая за конечное время преобразует начальные (входные) в заключительные (выходные) значения.

Определение 2. Для детерминированных вычислительных программ используют два вида логических спецификаций, которые называются:

- условиями частичной корректности,
- условиями тотальной корректности.

Эти условия имеют вид $\{\phi\}\pi\{\psi\}$ и $[\phi]\pi[\psi]$ соответственно, где π – программа, предусловие ϕ – это условие на входные данные, которое требуется перед исполнением программы π , а постусловие ψ – это условие на выходные данные, гарантируемое после исполнения программы π .

Определение 3. Говорят, что условие частичной корректности $\{\phi\}\pi\{\psi\}$ истинно (верно), или что программа π частично корректна по отношению к предусловию ϕ и постусловию ψ (обозначение $\models\{\phi\}\pi\{\psi\}$), если на любых входных данных, которые удовлетворяют свойству ϕ , программа π или не останавливается (зацикливается, зависает и тому подобное), или останавливается с выходными данными, которые удовлетворяют свойству ψ .

Определение 4. Говорят, что условие тотальной корректности $[\phi]\pi[\psi]$ истинно (верно), или что программа π тотально корректна по отношению к предусловию ϕ и постусловию ψ (обозначение $\models[\phi]\pi[\psi]$), если на любых входных данных, которые удовлетворяют свойству ϕ , программа π всегда останавливается с выходными данными, удовлетворяющими свойству ψ .

Таким образом, для детерминированных программ разница между частичной и тотальной корректностью состоит в завершаемости программы: частичная корректность допускает неостановку программы, а тотальная – запрещает. Например, пусть LOOP – это детерминированный бесконечный цикл WHILE TRUE DO $x := x + 1$, где x – целая переменная. Тогда $\models\{\text{TRUE}\}\text{LOOP}\{\text{TRUE}\}$, но неверно $\models[\text{TRUE}]\text{LOOP}[\text{TRUE}]$.

Пример спецификаций корректности: задача на вычисление целой части квадратного корня из натурального числа. Неформально программу, которая решает эту задачу, можно специфицировать так: по целому неотрицательному числу n вычислить $\lfloor n^{1/2} \rfloor$. В качестве решения программист написал следующую программу SR, которая (по мнению этого программиста) решает поставленную задачу (причём без использования операции умножения и без вложенных циклов):

```
VAR x,y,n : INT;  
x := 0 ; y := 0 ;
```

WHILE $y \leq n$ DO ($y := y + x + x + 1$; $x := x + 1$);
 $x := x - 1$.

Потом этот программист специфицировал программу условием частичной корректности $\{n \geq 0\}SR\{x^2 \leq n \ \& \ (x+1)^2 > n\}$. Истинность этого условия означает, что при любом натуральном начальном значении N переменной n эта программа SR если когда-либо завершит свою работу, то заключительное значение X переменной x будет таким, что $X^2 \leq N$ & $(X+1)^2 > N$, т. е. будет равно $[N^{1/2}]$. А истинность условия тотальной корректности $[n \geq 0]SR[x^2 \leq n \ \& \ (x+1)^2 > n]$ означает дополнительно, что программа SR обязательно завершит свою работу.

Определение 5. Резидентная программа — это программа, которая должна работать бесконечно долго, но может в процессе многократно получать входные значения и многократно выдавать выходные значения.

Определение 6. Для резидентных программ наиболее важны свойства следующих трёх типов:

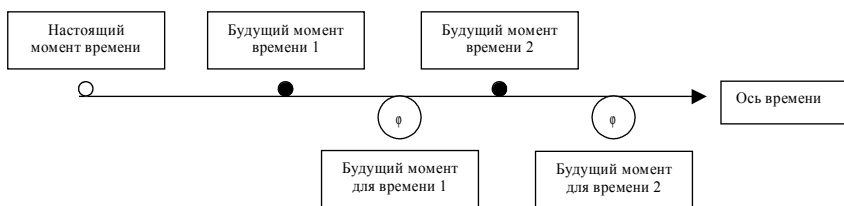
- прогресса¹⁴,
- безопасности,
- справедливости.

Свойства прогресса записываются в виде $\Diamond_{\pi}\phi$, свойства безопасности — в виде $\pi\phi$, а свойства справедливости — в виде $\pi\Diamond_{\pi}\phi$, где π - программа, а ϕ -свойство.

Определение 7. Для детерминированной резидентной программы π :

- свойство прогресса $\Diamond_{\pi}\phi$ означает, что рано или поздно, но будет такой момент, когда во время исполнения π наступит свойство ϕ ;
- свойство безопасности $\pi\phi$ означает, что во все моменты времени в будущем будет иметь место свойство ϕ ;
- свойство справедливости $\pi\Diamond_{\pi}\phi$ означает, что в будущем будет бесконечно часто выполняться свойство ϕ , т. е. в любой будущий момент времени (π) найдётся будущий момент времени (\Diamond_{π}), когда будет иметь место свойство ϕ .

¹⁴ Синоним: живости.



Пример спецификаций прогресса и безопасности: задача о роботах на Марсе. На поверхности Марса внутри небольшого кратера работают несколько автономных самоходных роботов, а на марсианской орбите находится орбитальная управляющая станция. Каждый из роботов оборудован приборами зрения, связи и компьютером. В свою очередь, орбитальная станция имеет приборы зрения, связи и бортовой компьютер. В случае появления у роботов внешних повреждений, протокол MR позволяет каждому роботу с внешними повреждениями узнать о их наличии при следующих предположениях об оборудовании станции и роботов.

Приборы зрения позволяют каждому из роботов видеть сразу всех остальных роботов, но не позволяют видеть себя самого. (В частности, каждый робот может видеть внешние повреждения любого другого робота, но не может видеть своих внешних повреждений.) Приборы связи позволяют каждому из роботов осуществлять надёжную связь с орбитальной станцией, когда она находится над горизонтом, но не позволяют общаться с остальными роботами (например, каждый робот имеет только свою радиочастоту).

Приборы зрения станции позволяют видеть всех роботов вместе (но не каждого в отдельности) когда станция находится над тем полушарием, в котором находятся роботы. (В частности, станция может видеть, когда среди роботов на поверхности Марса есть роботы с внешними повреждениями.) Приборы связи станции позволяют посылать сигналы всем роботам одновременно и получать сигналы от каждого робота в отдельности, когда станция находится над полушарием с роботами.

Компьютеры станции и каждого робота могут производить любые вычисления и абсолютно надёжны. Однако компьютеры роботов являются серверами, а компьютер орбитальной станции – клиентом, т. е. станция может посылать команды роботам, а те обязаны принимать эти команды к исполнению.

Описание протокола MR:

MR _{orb}	MR _{rob}
<pre> IF see("defective robots") THEN BEGIN send("there are defective") ; DO make(turn around Mars) ; send("report iff you are defective") UNTIL receive(any_reply) END </pre>	<pre> IF receive("there are defective") THEN BEGIN count number N of defective robots that you see ; FOR I:=1 TO N DO skip(a session with the orbiter) ; IF receive("report iff you are defective") THEN (reply("I am defective") ; DEF:=TRUE) ELSE DEF:=FALSE END </pre>

На неформальном уровне спецификация протокола MR может быть представлена следующим образом:

- всякий робот с внешними дефектами рано или поздно оповестит об этом станцию;
- всякий робот без внешних дефектов никогда не пошлёт станции сообщение о повреждениях.

В соответствии со смыслом условий прогресса и безопасности, эти неформальные спецификации можно представить следующим образом:

- $\text{defective-}i \rightarrow \Diamond_{MR} \text{report-}i$;
- $\neg \text{defective-}i \rightarrow {}_{MR} \neg \text{report-}i$,

где $\text{defective-}i$ и $\text{report-}i$ – предикаты, соответствующие роботу с номером i , означающие наличие внешних повреждений и посылку рапорта на станцию.

Эти примеры хорошо объясняют мнемонику названий свойств прогресса и безопасности:

- свойство прогресса ($\text{defective-}i \rightarrow \Diamond_{MR} \text{report-}i$) утверждает, что что-то правильное ($\text{report-}i$) для повреждённого робота ($\text{defective-}i$) наступит в будущем (\Diamond_{MR});
- свойство безопасности ($\neg \text{defective-}i \rightarrow {}_{MR} \neg \text{report-}i$) утверждает, что ничего неправильного ($\text{report-}i$) для целого робота ($\neg \text{defective-}i$) в будущем не наступит (${}_{MR}$).

Пример спецификации справедливости: планировщик времени центрального процессора. На однопроцессорном компьютере установлено несколько резидентных программ P_1, \dots, P_N как прикладных, так и системных, каждая из которых время от времени должна обрабатываться центральным процессором. Разделения времени центрального процессора между этими программами осуществляет планировщик SC (который сам является одной

из резидентных программ). Планировщик обязан обеспечить справедливый доступ к центральному процессору всех резидентных программ, т. е. гарантировать, что ни одна из программ P_1, \dots, P_N не ждёт бесконечно долго обработки центральным процессором, или, другими словами, что любая из резидентных программ обрабатывается бесконечно часто. В соответствии со смыслом условий справедливости эту неформальную спецификацию планировщика можно представить следующим образом: $\&_{i \in [1..N]} (sc \diamond_{sc} CPU(P_i))$.

Замечания по структуре материала курса

По уровню формализации материал данного курса можно разделить на две части: неформализованный, в котором учащиеся должны уяснить понятия, и формализованный, который надо знать и/или уметь.

Пример неформализованного материала – это понятие языка программирования. С этим понятием мы будем знакомиться на примере НеМо. Но то, что заложено в этот язык, не исчерпывает и не покрывает всего разнообразия ни общих парадигм языков программирования, ни их специфических особенностей. По-видимому, самое общее определение языка программирования заключается в следующем. Язык программирования – это язык для записи алгоритмов, предназначенных для реализации на компьютере. Уровень языка программирования – то, насколько язык ориентирован на понимание человеком сути алгоритмов, запрограммированных на этом языке. Поэтому два крайних полюса языков программирования – это автокод (ассемблер) конкретной ЭВМ (компьютера) и так называемый «псевдокод» – по программистки структурированное описание алгоритма в естественных терминах, реализуемость которого не вызывает сомнений у специалистов.

В свою очередь, формализованный материал будет также делиться на две части: что надо знать и что надо уметь. Знать надо формальные определения, утверждения и теоремы. Уметь – описывать, доказывать и реализовывать алгоритмы. Разумеется, знание утверждений и теорем предполагает умение их доказывать, а умение описывать, доказывать, и реализовывать алгоритмы предполагает знание соответствующей теории.

В курсе есть два вида определений: общие и частные. Первые носят общезначимый характер и будут выделены в тексте ключевым словом **Определение** (с номером), набранном жирным шрифтом. Эти определения достаточно лаконичны и будут выноситься в отдельные абзацы. Вторые – «привязаны» к языку НеМо, имеют смысл только в рамках данного курса и (как правило) будут выделяться посредством ключевого слова «Определение», набранного обычным шрифтом. С примерами определений первого вида мы уже встречались: определения условий частичной и тотальной кор-

ректности, условий прогресса, безопасности и справедливости – это примеры общезначимых определений.

Утверждения и теоремы будут состоять из формулировок и доказательств. Начинается утверждение или теорема словом **Утверждение** и **Теорема** (с номером) и завершаться знаком «■». Разделителем между формулировкой и доказательством служит слово **Доказательство**. Разница между утверждениями и теоремами состоит в том, что утверждения посвящены свойствам языка НеМо и тем самым «привязаны» к данному курсу, в то время как теоремы носят общий характер и не привязаны к выбранному в курсе варианту изложения учебного материала.

Все **Определения**, **Утверждения** и **Теоремы** в курсе нумеруются в пределах каждой лекции, и при ссылке на конкретное определение, утверждение или теорему мы будем писать, например, «утверждение ... из лекции ...».

В курсе есть два вида алгоритмов: общезначимые и конкретные. Первые носят общезначимый характер и будут выделены в тексте ключевым словом **Алгоритм**, набранным жирным шрифтом. Вторые – «привязаны» к языку НеМо, имеют смысл только в рамках данного курса и будут выделяться ключевым словом «Алгоритм», набранным обычным шрифтом.

Все алгоритмы, изложенные в данном курсе или предложенные для самостоятельной разработки и/или программной реализации в контрольных лабораторных работах, делится на два класса (наподобие программ на языке НеМо): вычислительные алгоритмы и алгоритмические процессы. Первые из них предназначены для получения однократного конечного результата (как и вычислительные программы), а вторые – на получение потенциально бесконечной последовательности «результатов» с учётом, возможно, новых входных данных (как и резидентные программы).

Для записи всех алгоритмов мы будем использовать императивный псевдокод. Главный механизм преобразования данных при таком подходе – явные операторы присваивания без побочных эффектов, в правых частях которых можно использовать рекурсивные функции. Механизм передачи параметров – по значению, ключевое слово для вычисленного значения функции – «возвращает значение». В качестве управляющих конструкций разрешается использовать стандартные IF-THEN-ELSE, WHILE-DO, FOR-DO и т. д. с единственным ограничением на отсутствие побочных эффектов во время проверки условий. Комментарий может располагаться в любом месте алгоритма начинается с символов «//», занимать несколько строчек, а заканчиваться переводом строки.

Все алгоритмы специфицируются с использованием пред- и постусловий, условий частичной и тотальной корректности, условий прогресса и безопасности, условий справедливости. Так, все ограничения на входные данные представляются в виде предусловий. В тех случаях, когда нас будет интересовать однократный результат работы алгоритма, то его свойства будут специфицироваться постусловием. Обычно пред- и постусловия записываются с использованием формализмов для условий частичной и тотальной корректности. Если мы захотим наложить некоторые ограничения на работу алгоритма, то будем использовать условия прогресса, безопасности и справедливости, записывая их на естественном языке.

Лекция 2. Неформальное введение в формальную верификацию

Методы спецификации и доказательства вычислительных программ

Вычислительные программы специфицируются условиями частичной и тотальной корректности. Для доказательства специфицированных вычислительных программ существуют два метода, разработанных Р. Флойдом в 1950-1960-х гг для так называемых детерминированных программ, у которых для каждого набора входных значений переменных возможно только одно «исполнение».

Метод Р. Флойда доказательства условий частичной корректности состоит в следующем:

$\{ \{\varphi\} \pi \{\psi\} \}$ – условие частичной корректности детерминированной программы }

// Предусловие метода.

- Представить программу π графически в виде блок-схемы и выбрать множество контрольных точек (на дугах), включающее начало и конец программы, такое, чтобы любой цикл по графу блок-схемы содержал контрольную точку.
- Сопоставить началу программы предусловие φ , концу программы – постусловие ψ , а каждой из оставшихся контрольных точек – некоторое условие. // Аннотация «инвариантами».
- Построить множество «участков» L , состоящее из всех невырожденных ациклических маршрутов по графу блок-схемы программы между всеми парами контрольных точек¹⁵.

¹⁵ Т. е. «участок» – это любой из ациклических маршрутов по графу блок-схемы программы, состоящий хотя бы из одной дуги, между любой парой контрольных точек (возможно совпадающих).

- Для каждого участка из L доказать: если условие, сопоставленное началу участка, верно перед началом исполнения участка, то условие, сопоставленное концу участка, верно после исполнения этого участка.

// Доказательство инвариантов.

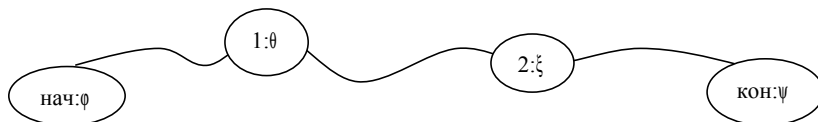
$\models \{\phi\} \pi \{\psi\}$, т. е. программа π частично корректна по отношению к предусловию ϕ и постусловию ψ // Постусловие метода.

Заметим, что метод Р. Флойда для условий частичной корректности не является алгоритмом, так как необходимо придумать и доказать инварианты (шаги 2 и 4).

Корректность метода можно обосновать следующим образом. Пусть ϕ имеет место перед исполнением программы π . Если при этом детерминированная программа π не останавливается, то автоматически $\models \{\phi\} \pi \{\psi\}$. Если же π останавливается, то пусть



какое-либо исполнение программы π . Выделим в нём все контрольные точки, через которые проходило это исполнение (на рисунке для определённости их всего 2), и выпишем условия, соответствующие всем этим контрольным точкам:



Участки этого исполнения:

- от начала до точки 1;
- от точки 1 до точки 2;
- от точки 2 до конца

являются ациклическими в силу шага 1 метода. В силу шага 4 метода, если

- ϕ верно в точке начала, то θ верно в точке 1;
- θ верно в точке 1, то ξ верно в точке 2;
- ξ верно в точке 2, то ψ верно в точке конца.

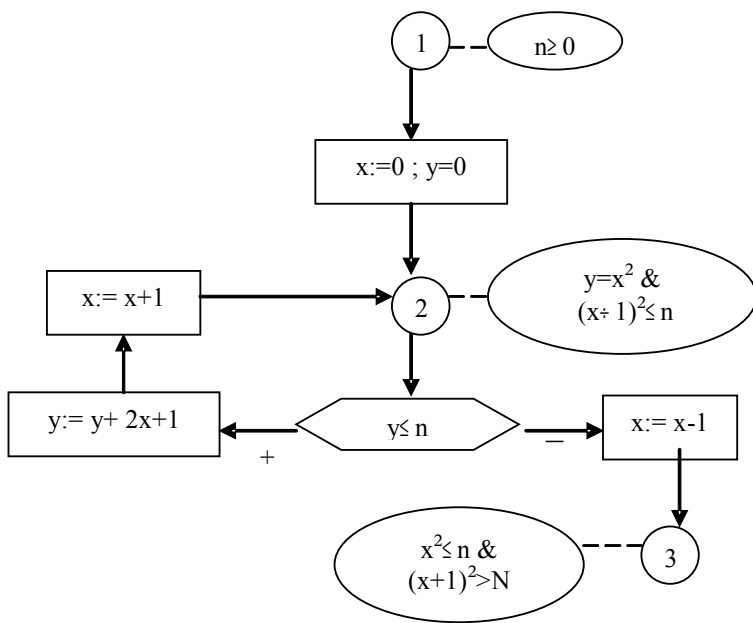
Следовательно, ψ имеет место после исполнения программы π , т. е. $\models \{\phi\} \pi \{\psi\}$ даже если детерминированная программа π завершает работу.

Пример применения метода: задача на вычисление целой части квадратного корня из натурального числа. В лекции 1 предложена следующая программа SR, решающая задачу

```
VAR x,y,n : INT;  
x := 0; y := 0;  
WHILE y <= n DO  
    (y := y + x + x + 1; x := x + 1);  
x := x - 1;
```

а также условие частичной корректности $\{n \geq 0\}SR\{x^2 \leq n \ \& \ (x + 1)^2 > n\}$. Очевидно, что SR – детерминированная программа. Поэтому предусловие метода Флойда выполнено, и мы можем применить сам метод.

1. Представим программу SR графически в виде блок-схемы и выберем множество контрольных точек 1, 2 и 3, включающее начало и конец программы, такое, что бы любой цикл по графу содержал некоторую из контрольных точек: Сопоставим начальной точке 1 предусловие ($n \geq 0$), конечной точке 3 – постусловие ($x^2 \leq n \ \& \ (x + 1)^2 > n$), а «внутренней» контрольной точке 2 – инвариант ($y = x^2 \ \& \ (x \div 1)^2 \leq n$), где « \div » – операция вычитания без перехода через 0 (т. е. $0 \div 1 = 0$).
2. Построим множество всех ациклических участков программы между всеми парами соседних контрольных точек $L = \{(1..2), (2+2), (2-3)\}$, где « $..$ » обозначает отсутствие условных операторов, а « $+$ » и « $-$ » – положительную и отрицательную ветвь условного оператора.



3. Для каждого ациклического участка из L докажем, что если условие, сопоставленное началу участка, верно перед началом исполнения участка, то условие, сопоставленное концу участка, верно после исполнения этого участка.

3.1. Участок (1..2). Если $n \geq 0$, то после присваиваний $x := 0$ и $y := 0$ очевидным образом $(y = x^2 \ \& \ (x + 1)^2 \leq n)$.

3.2. Участок (2+2). Пусть значение переменной n есть N , а значения переменных x и y перед исполнением этого участка есть X и Y . Пусть $(Y = X^2 \ \& \ (X + 1)^2 \leq N)$ перед исполнением участка. После исполнения присваивания $y := y + x + x + 1$ значение переменной y станет $(X + 1)^2$, а после исполнения присваивания $x := x + 1$ значение переменной x станет $(X + 1)$. Следовательно, новые значения переменных x и y после исполнения участка опять удовлетворяют соотношению $y = x^2$. Кроме того, так как участок начинается с положительной ветви условия $y \leq n$, то $Y \leq N$ и, следовательно, $X^2 \leq N$. Так как значение переменной x после исполнения участка есть $(X + 1)$, то, следовательно, это значение удовлетворяет неравенству $(x - 1)^2 \leq n$.

3.3. Участок (2–3). Пусть значение переменной n есть N , а значения переменных x и y перед исполнением этого участка есть X и Y . Пусть $(Y = X^2 \ \& \ (X + 1)^2 \leq N)$ перед исполнением участка. После исполнения присваивания $x := x - 1$ значение переменной x станет $(X + 1)$. Поэтому после исполнения этого участка $x^2 \leq n$. Кроме того, так как участок начинается с отрицательной ветви условия $y \leq n$, то $y > n$ и, следовательно, $X^2 > n$. Так как в конце этого участка значение переменной x равно $(X - 1)$, то, следовательно, после исполнения участка $(x + 1)^2 > n$.

Таким образом, $\models \{n \geq 0\} SR \{x^2 \leq n \ \& \ (x + 1)^2 > n\}$ – верное утверждение частичной корректности.

Метод Р. Флойда для доказательства условий тотальной корректности называется методом потенциалов, он применим только к детерминированным программам. Частный случай этого метода, который используется для доказательства завершаемости детерминированных программ, кратко можно описать следующим образом:

- выбирается конечное множество числовых значений, которые называются потенциалами;
- каждому возможному набору значений используемых переменных сопоставляется некоторый потенциал;

тогда если каждое исполнение тела любого цикла в алгоритме уменьшает значение потенциала, то алгоритм завершает работу.

Более строго метод потенциалов в общем случае можно описать следующим образом:

$\{ [\varphi] \pi [\psi] \}$ – условие тотальной корректности детерминированной программы } // Предусловие метода.

1. Применить к условию частичной корректности $\{ \varphi \} \pi \{ \psi \}$ метод Флойда.
2. Построить такое частично-упорядоченное множество «потенциалов» (D, \leq) и отображение f из множества конфигураций программы¹⁶ π в D , что:
 - 2.1. (D, \leq) является фундированным, т. е. в D нет бесконечных строго убывающих (по отношению \leq) цепей;
 - 2.2. Если начальные данные удовлетворяют предусловию φ , то однократное исполнение любого циклического участка в π уменьшает значение функции f , т. е. для любого циклического маршрута lp по блок-схеме программы, для любых конфигурации $c1$ и $c2$, если lp

¹⁶ Конфигурация программы – это набор одновременных значений всех переменных программы и место в блок-схеме, в котором находится управление.

начинается в конфигурации $c1$, а заканчивается в конфигурации $c2$, то $f(c1) > f(c2)$.

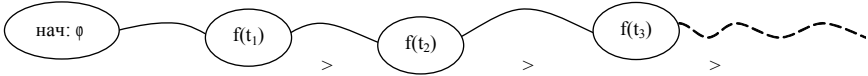
$\{ \models [\phi] \pi [\psi] \}$, т. е. программа π тотально корректна по отношению к предусловию ϕ и постусловию ψ // Постусловие метода.

Заметим, что метод Р. Флойда для условий тотальной корректности тоже не является алгоритмом, так как необходимо придумать и доказать инварианты, фундированное множество потенциалов, отображение в это множество, доказать, что потенциалы убывают в результате выполнения циклов (шаг 2).

Корректность метода потенциалов можно обосновать следующим образом. После шага 1 имеем $\models \{ \phi \} \pi \{ \psi \}$. Пусть шаг 2 также успешно завершился. Предположим, что ϕ имеет место перед исполнением программы π . Если при этом детерминированная программа π не останавливается, то её единственное исполнение является бесконечным и, следовательно, содержит бесконечное множество вхождений t_1, t_2, t_3, \dots некоторой из её контрольных точек:



Любой участок между двумя соседними из этих вхождений (включая эти точки) является циклом. Так как ϕ выполнено для начальных данных, то значения функции f в этих точках образуют бесконечную убывающую последовательность:



Существование этой бесконечной строго убывающей последовательности противоречит выбору фундированного множества (D, \leq) . Следовательно, предусловие ϕ гарантирует завершаемость исполнения программы π . Для детерминированной программы π завершаемость вместе с условием $\models \{ \phi \} \pi \{ \psi \}$ даёт $\models [\phi] \pi [\psi]$.

Пример применения метода: задача на вычисление целой части квадратного корня из натурального числа. Пусть $[n \geq 0]SR[x^2 \leq n \ \& \ (x+1)^2 > n]$ – условие тотальной корректности, где SR – программа, уже описанная выше в примере доказательства частичной корректности.

1. Выше был приведён пример применения метода Р. Флойда для доказательства условия частичной корректности $\{n \geq 0\}SR\{x^2 \leq n \ \& \ (x + 1)^2 > n\}$.
2. Тогда в качестве фундированного множества (D, \leq) примем множество всех натуральных чисел \mathbb{N} с естественным порядком на них, а в качестве отображения f из множества конфигураций программы SR в \mathbb{N} – отображение, которое сопоставляет каждой паре значений $N, Y \in \mathbb{N}$ переменных n и y программы целое число $(N + 1 \div Y)$. Исполнение цикла в программе SR возможно только при условии, что значение Y не превосходит значения N . Поэтому, перед исполнением цикла значение функции f – некоторое положительное число, но не 0. Во время исполнения тела цикла значение переменной n не изменяется, а переменной y увеличивается. Следовательно, значение функции f после исполнения цикла уменьшается.

Согласно методу потенциалов, $\models [n \geq 0]SR[x^2 \leq n \ \& \ (x + 1)^2 > n]$.

Пример Э. Дейкстры применения метода Р. Флойда

Проблема начальника порта – это следующая олимпиадная задача по программированию:

На рейде порта с N причалами находится ровно N кораблей. Начальник порта получает сообщение о штормовом предупреждении и должен назначить каждому из кораблей его определенный (индивидуальный) причал. Положения всех кораблей в момент получения штормового предупреждения известны. Положения причалов фиксированы и тоже известны. Непосредственно в этот момент никакое препятствие (особенности береговой линии, расположение других причалов или положение других кораблей на рейде) не мешает любому из кораблей двигаться прямо к любому из причалов. Но во избежание столкновений маршруты кораблей не должны пересекаться. Помогите начальнику порта распределить корабли по причалам.

Геометрическая модель проблемы – это N чёрных и N белых точек на плоскости, соответствующих положениям кораблей на рейде и расположению причалов, а возможные (гипотетические) маршруты – отрезки прямых между белыми и чёрными точками. Заметим, кстати, что по условию задачи никакие три из этих точек не являются коллинеарными (т. е. не лежат на одной прямой). Нам необходимо построить (если это вообще возможно) N попарно непересекающихся отрезков, у каждого из которых один конец – это некоторая белая точка, а другой – некоторая чёрная точка.

Целевой «объект» геометрической модели проблемы начальника порта – множество из N отрезков с концами (разного цвета) в заданных N чёрных и N белых точках, причём множество, не содержащее пересекающихся отрезков. В связи с этим имеет смысл ввести специальный тип данных СОЕДинение, значения которого – всевозможные множества из N отрезков с концами (разного цвета) в заданных N чёрных и N белых точках. Среди всех значений типа СОЕД нас интересуют такие множества отрезков, которые не содержат пересечений, поэтому кажется вполне оправданным ввести булевскую функцию $XOP(X: \text{СОЕД}) : \text{BOOL}$ на значениях этого типа, которая возвращает значение TRUE тогда и только тогда, когда множество отрезков X не содержит пересекающихся отрезков. Заметим, что тип СОЕД состоит из $N!$ различных значений: первая белая точка может быть соединена с любой из N чёрных точек, вторая белая – с любой из оставшихся $(N-1)$ чёрных точек и т. д. Поэтому можно считать, что перед нами перечислимый тип, среди значений которого есть первый элемент, доступный благодаря константе ПЕРВ: СОЕД, и есть функция следования СЛЕД($X: \text{СОЕД}$) : СОЕД, которая по каждому элементу (кроме последнего), переданному в качестве значения фактического параметра, возвращает следующий элемент.

В терминах типа данных СОЕД геометрическая модель проблемы начальника порта интерпретируется следующим простым способом: среди всех возможных значений типа СОЕД найти (если есть) такой, который удовлетворяет XOP . Эта переформулировка позволяет спроектировать первый чисто переборный алгоритм решения проблемы начальника порта¹⁷:

```
VAR X: СОЕД; VAR C, N: INT;
X := ПЕРВ; C := 0;
WHILE C < N! & ¬XOP(X) DO X := СЛЕД(X); C := C + 1 OD;
IF C = N! THEN OUTPUT ('SORRY!') ELSE OUTPUT (X) .
```

(Роль счётчика C в этом алгоритме очевидна: он подсчитывает, сколько соединений из $N!$ возможных уже проверено, и позволяет завершить цикл, как только все соединения будут проверены и ни одного хорошего из них не будет найдено.)

Достоинство такого алгоритма – простота описания, а главный недостаток – высокая верхняя оценка сложности (так как в худшем случае не исключён перебор всех возможных значений типа СОЕД). Отметим, однако, что, во-первых, данный алгоритм никак не использует тот факт, что ника-

¹⁷ Предупреждение для поклонников нотации языка Си: здесь и далее знак «!» обозначает факториал, а не операцию отрицания.

кие три точки не являются коллинеарными, а, во-вторых, данный алгоритм найдёт хорошее соединение тогда и только тогда, когда оно существует¹⁸.

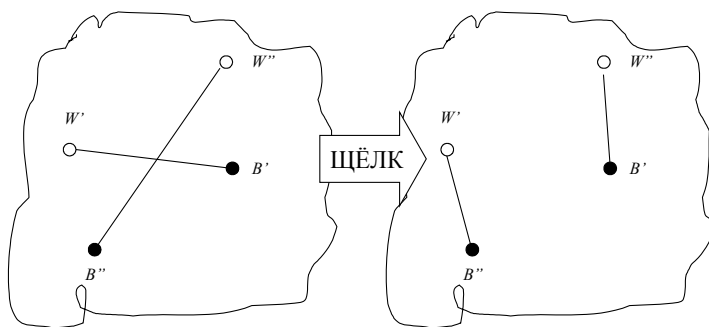
Отнюдь не оптимистические результаты анализа эффективности (пропорциональной $N!$), полноты использования информации (не используется отсутствие коллинеарных точек) и результативности (ответ 'SORRY!' возможен) наталкивают на мысль попробовать применить какую-нибудь эвристику. В данном случае речь идёт о локальном устранении пересечений в множестве из N отрезков с разными концами в чёрных и белых точках, в надежде, что это позволит устранить в конце концов все пересечения. В связи с этим имеет смысл использовать тот же специальный тип данных СОЕД, что и в алгоритме полного перебора, с той же булевой функцией $\text{ХОП}(X: \text{СОЕД}) : \text{BOOL}$, среди значений которого есть первый элемент, доступный благодаря константе ПЕРВ: СОЕД. Но теперь вместо функции следования $\text{СЛЕД}(X: \text{СОЕД}) : \text{СОЕД}$, позволяющей осуществить перебор все значения этого типа, будет использоваться функция $\text{ЩЁЛК}(X: \text{СОЕД}) : \text{СОЕД}$, которая реализует желаемую эвристику локально устраняя пересечения отрезков, т. е. выбирает некоторую пару (если такая существует) пересекающихся отрезков $[B', W'] \cap [B'', W''] \neq \emptyset$ из множества отрезков X и «переЩёлкивает» их, т.е. заменяет на пару отрезков $[B', W'']$ и $[B'', W']$, т. е. возвращает множество отрезков $(X \setminus \{[B', W'], [B'', W'']\}) \cup \{[B', W''], [B'', W']\}$, как это показано на следующем рисунке.

Теперь мы можем описать алгоритм поиска с эвристикой для проблемы начальника порта:

```
VAR X: СОЕД; VAR C, N: INT;
X := ПЕРВ ; C := 0 ;
WHILE C < N! & ¬ХОП(X) DO X := ЩЁЛК(X); C := C+1 OD ;
IF C = N! THEN OUTPUT ('SORRY!') ELSE OUTPUT (X)
```

Фактически этот алгоритм эвристического поиска повторяет алгоритм полного перебора, но использует вместо функции следования, которая перебирает все возможные соединения, функцию локального устранения конфликта, которая перебирает только некоторые соединения, получаемые из одного начального в результате перещёлкивания отрезков. Значения счётчика C в этом алгоритме уже не позволяют отслеживать, что проверены все соединения, но позволяют избегать закливания, поскольку всех соединений ровно $N!$, и, следовательно, период функции ЩЁЛК не более $N!$

¹⁸ Другими словами заранее не известно, остановится ли алгоритм с ответом в виде конкретного множества из N отрезков между чёрными и белыми точками без пересечений или остановится с неутешительным ответом 'SORRY!'



Достоинства эвристического алгоритма – простота описания и правдоподобная эвристика, его недостатки – с одной стороны, высокая оценка сложности (так как не исключён перебор всех возможных значений типа СОЕД), а с другой – возможная «неполнота», т. е. алгоритм может не найти хорошее соединение даже тогда, когда оно существует (не исключено, что функция ЩЁЛК не перебирает все значения типа СОЕД, а «циклит» по значениям, которые содержат пересекающиеся отрезки). Отметим также в очередной раз, что данный алгоритм пока никак не использует тот факт, что никакие три точки не являются коллинеарными.

Возможная неполнота эвристического алгоритма вызывает желание исследовать, когда всё-таки функция ЩЁЛК не закидывается на значениях типа СОЕД, содержащих пересекающиеся отрезки. Другими словами, когда следующий вариант эвристического алгоритма без использования счётчика всё-таки решает проблему начальника порта:

```

VAR X: СОЕД;
X := ПЕРВ ;
WHILE ¬ХОП(X) DO X := ЩЁЛК(X) OD ;
OUTPUT (X)

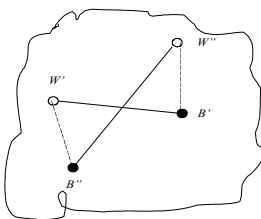
```

Этот вариант эвристического алгоритма решения проблемы начальника порта предложил в 1994 г. в публичной лекции классик науки программи-

рования Э. Дейкстра¹⁹. Отметим очевидный факт: если этот алгоритм завершает работу, то он выдаёт решение проблемы начальника порта. Удивительным является то, что этот алгоритм гарантированно останавливается и, следовательно, является полным алгоритмом решения проблемы начальника порта! Для доказательства завершаемости (т. е. полноты) своего эвристического алгоритма Э. Дейкстра применил как раз метод потенциалов Флойда.

Применительно к своему эвристическому алгоритму Э. Дейкстра рассуждал следующим образом. Для любого значения X типа СОЕД примем в качестве его потенциала $P(X)$ сумму длин отрезков, которые входят в X : $P(X) = \sum_{[B,W] \in X} |B,W|$. Поскольку множество значений типа СОЕД конечно, то и соответствующее множество потенциалов тоже конечно. Тело единственного цикла эвристического алгоритма состоит из единственного оператора присваивания $X := \text{ЩЁЛК}(X)$, а исполнение этого тела уменьшает значения потенциала, что явствует из следующих рассуждений (см. рисунок): в силу неравенства треугольника²⁰ $|B'W''| + |B'',W'| < |B'W'| + |B',W''|$ и поэтому

$$\begin{aligned} P(X) &= \\ &= (\sum_{[B,W] \in X} |B,W|) = (|B',W'| + |B'',W''|) + (\sum_{[B,W] \in X \setminus \{[B',W'], [B'',W'']\}} |B,W|) > \\ &> (|B',W''| + |B'',W'|) + (\sum_{[B,W] \in X \setminus \{[B',W'], [B'',W'']\}} |B,W|) = (\sum_{[B,W] \in \text{ЩЁЛК}(X)} |B,W|) = \\ &= P(\text{ЩЁЛК}(X)). \end{aligned}$$



¹⁹ К сожалению, эта лекция не опубликована в электронном архиве Э. Дейкстры на <http://www.cs.utexas.edu/users/EWD/>.

²⁰ Неравенство треугольника применимо, ибо в соответствии с геометрической моделью проблемы начальника порта ни какие три точки не лежат на одной прямой.

Следовательно, в соответствии с методом потенциалов Флойда эвристический алгоритм всегда останавливается и является полным алгоритмом решения проблемы начальника порта.

Достоинства эвристического алгоритма Э. Дейкстры – изящество представления и полнота решения исходной проблемы. Главным недостатком этого алгоритма остаётся высокая верхняя оценка сложности, так как по-прежнему не исключён перебор всех возможных значений типа СОЕД. Отметим также, что при обосновании полноты этого алгоритма явно используется тот факт, что никакие три точки не являются коллинеарными.

Методы спецификации и доказательства резидентных программ

Резидентные (реактивные) программы специфицируются условиями безопасности, прогресса и справедливости. Для них З. Манна и А. Пнуэли разработали в 1980-1990-е гг. так называемые «проблемно-ориентированные принципы доказательства».

Метод («принцип») З. Манна – А. Пнуэли для условий безопасности состоит в следующем:

- { $(\varphi \rightarrow \pi\psi)$ – условие безопасности программы } // Предусловие метода.
1. Показать, что $(\varphi \rightarrow \psi)$ имеет место в любой начальной конфигурации π .
 2. Показать, что для любой конфигурации программы π , которая достижима из какой-либо начальной конфигурации удовлетворяющей φ , если эта достижимая конфигурация обладает свойством ψ , то любая непосредственно следующая конфигурация также обладает свойством ψ .
- { Если перед началом выполнения программы π имеет место свойство φ , то свойство ψ имеет место всегда во время исполнения программы. }
- // Постусловие.

Обоснование принципа для условий безопасности. Пусть $\varphi \rightarrow \pi\psi$ – условие безопасности, для которого удалось применить принцип, и условие φ выполнено в начале какого-либо исполнения программы π . Предположим, что в этом исполнении есть конфигурация, в которой опровергается свойство ψ (т. е. которая не обладает этим свойством). Выберем первую такую конфигурацию. Выбранная конфигурация не является начальной, так как начальная конфигурация должна обладать свойством φ , которое (согласно шагу 1 принципа) гарантирует ψ . Значит, выбранной конфигурации непосредственно предшествует в данном исполнении некоторая другая конфигурация, которая (в силу принадлежности выбранному исполнению) достижима из начальной конфигурации, обладающей свойством φ и в то же время сама обладает свойством ψ (в силу предшествования первой конфигурации,

опровергающей ψ). Следовательно, в выбранной конфигурации также должно выполняться свойство ψ (согласно пункту 2 принципа). Значит, предположение о существовании в выбранном исполнении программы π конфигурации, в которой опровергается ψ , неверно. Таким образом, если перед началом выполнения программы π имеет место свойство ϕ , то свойство ψ имеет место всегда во время исполнения программы.

Метод («принцип») 3. Манны – А. Пнуэли для условий прогресса состоит в следующем:

- $\{ (\phi \rightarrow \Diamond_{\pi}\psi) - \text{условие прогресса программы} \} // \text{Предусловие метода.}$
- Построить такое фундированное множество (D, \leq) и отображение f из множества конфигураций программы π в D , что для любой конфигурации программы π , которая достижима из какой-либо начальной конфигурации, удовлетворяющей ϕ , если в этой достижимой конфигурации функция f принимает какое-либо неминимальное значение (т. е. не достигает минимума), то при любом возможном исполнении из этой конфигурации достижима другая конфигурация, в которой функция f принимает какое-либо меньшее значение.
- Показать, что любая конфигурация программы π , которая достижима из какой-либо начальной конфигурации, удовлетворяющей ϕ , и в которой функция f принимает какое-либо минимальное значение, обладает свойством ψ .
- $\{ \text{Если перед началом выполнения программы } \pi \text{ имеет место свойство } \phi, \text{ то свойство } \psi \text{ наступит во время любого исполнения программы.} \} // \text{Постусловие.}$

Обоснование принципа для условий прогресса. Пусть $\phi \rightarrow \Diamond_{\pi}\psi$ – условие прогресса, для которого удалось применить принцип, и условие ϕ выполнено в начале какого-либо исполнения программы π . Предположим, что в этом исполнении нет конфигурации, которая обладает свойством ψ . Пусть (D, \leq) и f – построенные на шаге 1 фундированное множество и отображение конфигураций программы π в это множество. Построим бесконечную последовательность конфигураций $c_0, \dots, c_i, c_{(i+1)}, \dots$, которые встречаются в выбранном исполнении, где функция f принимает значения $v_0 > \dots > v_i > v_{(i+1)} > \dots$. Пусть c_0 – это начальная конфигурация этого исполнения. В c_0 опровергается ψ , следовательно (согласно шагу 2 принципа), функция f не достигает минимума в этой конфигурации, а принимает некоторое значение v_0 , отличное от минимального. Предположим, что для некоторого $i \geq 0$ уже подобраны конфигурации c_0, \dots, c_i из выбранного исполнения, в которых f не достигает минимума и принимает значения $v_0 > \dots > v_i$. Так как c_i – кон-

фигурация, достижимая из начальной конфигурации, в которой имеет место ϕ , и значение $f(c_i) = v_i$ не является минимальным в (D, \leq) , то (согласно шагу 1 принципа) в выбранном исполнении найдётся конфигурация $c_{(i+1)}$, достижимая из c_i такая, что $f(c_{(i+1)}) < v_i = f(c_i)$. По сделанному предположению, в выбранном исполнении нет конфигурации, которая обладает свойством ψ . Поэтому в $c_{(i+1)}$ опровергается ψ , следовательно (согласно шагу 2 принципа) функция f не достигает минимума в этой конфигурации, а принимает некоторое значение $v_{(i+1)}$, отличное от минимального. Таким образом строится вся бесконечная последовательность конфигураций $c_0, \dots, c_i, c_{(i+1)}, \dots$, встречающихся в выбранном исполнении и в которых функция f принимает значения $v_0 > \dots > v_i > v_{(i+1)} > \dots$. Последнее противоречит фундированности множества (D, \leq) . Следовательно, неверно предположение об отсутствии в выбранном исполнении конфигурации, в которой выполняется свойство ψ .

Оба принципа 3. Манны – А. Пнуэли имеют явную аналогию с методами Р. Флойда доказательства частичной и тотальной корректности. Более того, методы Р. Флойда могут быть переформулированы как частные специальные случаи принципов 3. Манны – А. Пнуэли в силу того, что для любой детерминированной программы π , любых пред- и постусловий ϕ и ψ :

- условие частичной корректности $\{\phi\}\pi\{\psi\}$ равносильно условию безопасности $((\text{at начало}) \& \phi) \rightarrow_{\pi} ((\text{at конец}) \rightarrow \psi)$;
- условие тотальной корректности $[\phi]\pi[\psi]$ равносильно условию прогресса $((\text{at начало}) \& \phi) \rightarrow \Diamond_{\pi} ((\text{at конец}) \& \psi)$.

Пример применения принципов. В лекции 1 был представлен протокол MR, решающий задачу обнаружения внешних повреждений роботами, и спецификация его свойств прогресса и безопасности

- $\text{defective-}i \rightarrow \Diamond_{MR} \text{report-}i$;
 - $\neg \text{defective-}i \rightarrow_{MR} \neg \text{report-}i$
- для всех номеров роботов.

Применим принцип для условия прогресса $\text{defective-}i \rightarrow \Diamond_{MR} \text{report-}i$. Пусть $M > 0$ – общее число всех роботов с внешними дефектами. В качестве фундированного множества (D, \leq) примем отрезок натуральных чисел $[0..M]$ с обратным порядком²¹. В качестве функции f примем отображение, которое сопоставляет каждой конфигурации число сеансов связи с орбитальной станцией. Если в начале исполнения имеет место $\text{defective-}i$, то значение переменной N у робота с номером i равно $(M - 1)$. Поэтому в начальной и в $(M - 1)$ -ой следующих конфигурациях функция f не достигает

²¹ Прямой порядок на отрезке $[0..M]$ – это $0 < 1 < \dots < (M - 1) < M$, а обратный порядок на этом же отрезке – это $0 > 1 > \dots > (M - 1) > M$.

минимума, робот i не посылает подтверждение станции, а станция инициирует ещё один следующий сеанс связи, т. е. в следующей конфигурации функция f принимает меньшее значение в (D, \leq) . Однако в конфигурации, в которой функция f принимает минимальное значение (т. е. число сеансов связи со станцией равно M), выполняется последний оператор программы для робота i , и, следовательно, робот i посылает подтверждение станции, т.е. имеет место $\text{report-}i$.

Применим принцип для условия безопасности $\neg \text{defective-}i \rightarrow_{MR} \neg \text{report-}i$. Пусть опять $M > 0$ – общее число всех роботов с внешними дефектами. Если в начале исполнения имеет место $\neg \text{defective-}i$, то значение переменной N у робота с номером i равно M . Множество конфигураций распадается на три непересекающихся подмножества:

- начальная и первые $(M - 2)$ конфигураций;
- $(M - 1)$ -ая конфигурация;
- все остальные конфигурации начиная с M -ой.

В начальной и в следующих $(M - 2)$ конфигурациях ни один из роботов не посылает подтверждения станции, а станция инициирует в следующей конфигурации ещё один сеанс связи. В $(M - 1)$ -ой конфигурации все целые роботы молчат, а дефектные – отвечают. Поэтому во всех последующих конфигурациях (начиная с номера M) станция уже не посылает нового запроса, и, следовательно, ни один робот не отвечает. Таким образом, во всех конфигурациях имеет место $\neg \text{report-}i$ для всех целых роботов.

Лекция 3. Воспоминания о теории множеств и математической логике – I

Постулаты неформальной теории множества

Множество есть многое мыслимое как единое.

Г. Кантор

Концептуальные определения – удел философии, они мало пригодны для ученых и инженеров. Например, философ может рассуждать о компьютерах вообще, о том, что присуще всем компьютерам, что отличает компьютер от калькулятора и т. д. Тем самым он неявно выделяет их в отдельную совокупность. Но серьезный инженер или ученый в профессиональном разговоре не будет использовать «совокупность всех компьютеров» из-за неопределенности что входит, а что не входит в эту совокупность. В частности, совершенно не понятно, какое из перечисленных ниже «устройств» можно считать компьютером:

- счетные машины В. Оккама (XV в.) и Б. Паскаля (XVI в.);
- аналитическую машину Ч. Бэбиджа (XIX в.);
- счетные машины К. Цузе (1930–1945 гг.);
- Малую Электронно-Счетную Машину И. С. Лебедева (25 декабря 1951 г.);
- машины IBM серии 360/370 (1965–1980 гг.);
- RII 800/128/40Gd/... ?

Для того, чтобы преодолеть расплывчатость концептуального понятия «совокупность», математики постулируют аксиомы (неформальной) теории множеств. Эти аксиомы утверждают существование некоторых множеств и устанавливают отношения между множествами наподобие того, как постулирует существование королей и их власть над народом следующая «неформальная теория монархии»:

1. Существует хотя бы один монарх.
2. Каждый монарх властвует хотя бы над одним народом.
3. Никакой народ не подвластен сразу двум монархам.
4.

Неформальная теория множеств постулирует существование пустого множества, перечислимых множеств (т. е. заданных явным перечислением элементов), бесконечного множества, отображений (функций) из множества во множество, множества всех подмножеств множества, выделения подмножества по свойству элементов, объединения множеств, декартового произведения множеств. Эта теория может быть сформулирована с использованием отношения «быть элементом» (\in), «не быть элементом» (\notin), «быть подмножеством» (\subseteq), «равенство» элементов и множеств ($=$). Сформулируем постулаты (аксиомы) неформальной теории множеств на «человеческом» языке с использованием отношений « \in », « \notin », « \subseteq » и « $=$ ».

3. Существует пустое множество \emptyset , которое не имеет ни одного элемента: $A \notin \emptyset$ для любого множества A .
4. Для любой конечной совокупности множеств A_1, \dots, A_n существует множество $\{A_1, \dots, A_n\}$, элементами которого является в точности A_1, \dots, A_n .
5. Существует множество ω , которое содержит \emptyset и которое вместе со всяким своим элементом A содержит двухэлементное множество $\{A, \{A\}\}$: для любого A , если $A \in \omega$, то $\{A, \{A\}\} \in \omega$.

6. Для любых множеств A и B любая всюду определённая функция²² $F: A \rightarrow B$ является множеством.
7. Для любого множества A совокупность всех его подмножеств (обозначение 2^A или $\{ B : B \subseteq A \}$) является множеством.
8. Для любого множества A и любого свойства ϕ , которое удаётся сформулировать в терминах множеств и отношений между ними, можно выделить множество $\{ B \in A : \phi \}$ тех элементов A , которые обладают свойством ϕ .
9. Для любого множества A существует множество-объединение $(\cup_{B \in A} B)$, которое состоит из «элементов элементов» множества A : $C \in (\cup_{B \in A} B)$, если $C \in B$ для некоторого $B \in A$.
10. Для любых множеств A и B , любой всюду определённой функции $I: A \rightarrow B$ если $I(A)$ не содержит пустого множества (т. е. $\emptyset \notin I(A)$), то существует непустое множество-произведение $(\prod_{C \in A} I(C))$, состоящее из всех функций F , которые по всякому элементу C множества A возвращают значение $F(C) \in I(C)$ – «представителя» элемента $I(C) \in B$.
11. Для любых множеств A и B , B есть подмножество A , когда всякий элемент B входит в A : $B \subseteq A$ тогда и только тогда, когда для любого C из $C \in B$ следует $C \in A$.
12. Множества равны, когда они включают друг друга: $A = B$ тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$.

Множествами принято считать только те «объекты», которые удовлетворяют всем перечисленным аксиомам. Заметим, что далеко не всякий объект будет множеством. Например, «совокупность всех компьютеров» не является множеством хотя бы уже потому, что категория «компьютер» вообще отсутствует в этих аксиомах. Или другой пример: совокупность всех множеств, которые не содержат сами себя: $A \equiv \{ B : B \text{ есть множество, и } B \notin B \}$ не является множеством, но в этом случае такой простой аргумент, как в случае с совокупностью всех компьютеров, не проходит.

Для того, чтобы показать что $A \equiv \{ B : B \text{ есть множество и } B \notin B \}$ не является множеством, предположим противное: пусть A является множеством. Тогда $A \in A$ или $A \notin A$. В первом случае (по определению) $A \notin A$. Во втором (тоже по определению) $A \in A$. В обоих случаях получилось противоречие: $A \in A \Leftrightarrow A \notin A$. Значит A всё-таки не множество...

²² Пусть A и B – произвольные множества. Функция F из A в B (обозначение $F: A \rightarrow B$) – это произвольная совокупность упорядоченных пар, такая, что для любых $(C, D') \in F$ и $(C, D'') \in F$ имеем $C \in A$ и $D' = D'' \in B$. Функция $F: A \rightarrow B$ называется всюдуопределённой (на A), когда для любого $D \in A$ существует такой $D \in B$, что $(C, D) \in A$.

Из сказанного о совокупности A следует, что так называемое «множество всех множеств» $\text{Universum} \equiv \{ X : X - \text{множество} \}$ тоже не является множеством. В противном случае в соответствии с аксиомой выделения b совокупность A должна была бы быть множеством, так как $A = \{ B \in \text{Universum} : B \in B \}$.

Аксиомы неформальной теории множеств вводят только три операции на множествах: посторонние множества всех подмножеств, объединение и декартово произведение. Причём объединение и произведение пока определены в несколько необычном «формате» ($\cup_{B \in A} B$) и ($\prod_{B \in A} I(B)$), в то время как обычный «формат» - это $X \cup Y$ для объединения и $X \times Y$ для произведения двух множеств X и Y . Три другие важные и полезные операции пока остались неопределёнными: это пересечение ($X \cap Y$) и разность ($X \setminus Y$) множеств X и Y , а также множество Y^X всех отображений (функций) из X в Y . Но здесь дело обстоит просто, как это будет показано в следующем абзаце²³.

Пусть X и Y – два множества.

3. По аксиоме 2 существует множество $A = \{X, Y\}$. По аксиоме 7 существует множество ($\cup_{B \in A} B$), которое состоит из «элементов элементов» A , т. е. из элементов X и Y . Следовательно, множество ($\cup_{B \in A} B$) и есть обычное объединение ($X \cup Y$).
4. Если X или Y – пустое множество, то обычное декартово произведение $X \times Y$ – тоже пустое множество, его существование уже постулировано в первой аксиоме. Если же $X \neq \emptyset$ и $Y \neq \emptyset$, то обычное декартово произведение $X \times Y$ можно определить так. По аксиоме 3 существует бесконечное множество ω , которое содержит среди своих элементов \emptyset , $\{\emptyset\}$, $\{\{\emptyset\}\}$, $\{\{\emptyset, \{\emptyset\}\}\}$ и т. д. Обозначим эти элементы порядком $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$ и т. д.²⁴ По аксиоме 2 существуют два множества $A = \{\langle 1 \rangle, \langle 2 \rangle\}$ и $B = \{X, Y\}$. Пусть $I: A \rightarrow B$ – функция из A в B , такая, что $I(\langle 1 \rangle) = X$ и $I(\langle 2 \rangle) = Y$. По аксиоме 8 существует множество ($\prod_{C \in A} I(C)$), состоящее из всех «упорядоченных пар», т. е. функций, сопоставляющих $\langle 1 \rangle$ значение из $X = I(\langle 1 \rangle)$, а $\langle 2 \rangle$ – значение из $Y = I(\langle 2 \rangle)$.

²³ Если изложенный материал покажется сложным, то можно обратиться к следующим источникам для первоначального знакомства с неформальной теорией множеств:

- 1) Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л. Алгебра. Языки. Программирование. Киев: Наукова Думка, 1978. Гл. 1. Множества.
- 2) Кук Д. Бейз Г. Компьютерная математика. М.: Наука, 1990. Гл.1-3.

²⁴ Доказательство того, что ряд $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$, ... обладает свойствами натуральных чисел, выходит за пределы курса. Заинтересованного читателя можно отослать к специальной литературе по теории множеств и основаниям математики, например: Ершов Ю.Л., Палютин Е.А. Математическая логика. М.: Наука, 1987. Гл. 2. Теория множеств.

Следовательно, это множество $(\Pi_{C \in A} I(C))$ и есть обычное декартово произведение $(X \times Y)$.

5. По аксиоме 6 (выделения) существуют множества $\{C \in A : C \in B\}$ и $\{C \in A : C \notin B\}$. Это и есть обычные пересечение множеств $A \cap B$ и разность $A \setminus B$ соответственно.
6. Если X – пустое множество \emptyset , то множество всех отображений X в Y состоит из единственной пустой функции \emptyset (которая никакой аргумент переводит в никакой результат), и, следовательно, $Y^X = \{\emptyset\}$. Если же $X \neq \emptyset$, но Y – пустое множество \emptyset , то $Y^X = \emptyset$. Если же $X \neq \emptyset$ и $Y \neq \emptyset$, то определим тождественную функцию $I: X \rightarrow \{Y\}$, которая каждому $C \in X$ сопоставляет значение $I(C) = Y$. По аксиоме 8 существует множество $(\Pi_{C \in X} I(C))$, которое состоит из всех функций, сопоставляющих элементам $C \in X$ значения из $Y = I(C)$. Следовательно, это множество $(\Pi_{C \in X} I(C))$ и есть обычное множество всех отображений Y^X .

При изложении элементов неформальной теории множеств мы определили понятие функции как множества упорядоченных пар, в которых каждому аргументу (первой компоненте упорядоченной пары) соответствует единственный результат (вторая компонента упорядоченной пары). Для задания функций как соответствий на множествах обычно используют некоторую математическую нотацию. Например, функцию $f(x) = 2x+3$ на вещественных числах в традиционной математической нотации можно задать так: $f: \mathbb{R} \rightarrow \mathbb{R}$ где $f(x) = 2x+3$.

Однако наряду с этой общеизвестной нотацией сложилась и другая нотация, широко используемая в математической логике и теории программирования. Речь идёт о так называемой λ -нотации (читается «лямбда-нотация»). Суть её поясним на примере:

1. « $2x + 3$, где $x \in \mathbb{R}$ » – это число (но не функция), значение которого зависит от значения числового параметра x по закону $2x + 3$;
2. « $\lambda x \in \mathbb{R} . (2x + 3)$ » – это функция (без имени), где x объявлен переменной посредством спецификатора « λ » (а не параметром с каким-то значением), которая каждому числу $X \in \mathbb{R}$ сопоставляет число $(2X + 3) \in \mathbb{R}$;
3. « $f = \lambda x \in \mathbb{R} . (2x + 3)$ » – это высказывание о том, что функция « f » совпадает (как множество) с функцией $\lambda x \in \mathbb{R} . (2x + 3)$.

Разберём ещё один пример. Пусть $g = \lambda m \in \mathbb{N} . (\lambda n \in \mathbb{N} . (\lambda x \in \mathbb{R} . (m \times x + n)))$. Эта функция от одной переменной m , которая «пробегаёт» натуральные числа; она (функция) сопоставляет каждому натуральному числу M функцию $h_M = \lambda n \in \mathbb{N} . (\lambda x \in \mathbb{R} . (M \times x + n))$, в частности, $h_2 = \lambda n \in \mathbb{N} . (\lambda x \in \mathbb{R} . (2x + n))$. В свою очередь, h_M – тоже функция одной переменной n , которая

тоже «пробегают» натуральные числа и сопоставляет каждому натуральному числу N функцию $f_{M,N} = \lambda x \in \mathbf{R}. (M \times x + N)$, в частности $f_{2,3}$ – это уже знакомая нам функция $f = (\lambda x \in \mathbf{R}. (2x + 3))$.

Начала теории формальных языков²⁵

Всякий язык определяется своим синтаксисом, семантикой и прагматикой. Синтаксис – это «правописание» слов или фраз языка, семантика позволяет придать смысл правильно записанным выражениям, а прагматика определяет область применения правильно записанных и осмысленных фраз.

Однако формальный язык – это только множество слов, которые этому языку принадлежат, так сказать являются «правильными». Другими словами, формальный язык – это «голый» синтаксис без семантики и прагматики.

В программировании для определения формальных языков часто используется так называемая нотация Бэкуса – Наура. В этой нотации понятия служат для выделения частей правильных фраз (наподобие того, как понятия «подлежащее», «сказуемое» и т. д. используются в естественных языках). В нотации Бэкуса – Наура понятия заключаются в угловые скобки « \langle ,, \rangle ». Определения таких понятий записываются с использованием специального значка « $::=$ » следующим образом:

понятие $::=$ определение,

где в определении могут быть несколько альтернатив, перечисленных через знак-разделитель « \langle ». Каждое из альтернативных определений строится из цепочки понятий и неопределяемых (базовых или терминальных) символов. В определениях допускается рекурсия, т. е. одно понятие может определяться через другое, а то, в свою очередь, через первое понятие.

Пояснение как пользоваться этой нотацией на примере синтаксиса для языка свойств ϕ в аксиоме выделения ϕ неформальной теории множеств.

$\langle \text{пустое множество} \rangle ::= \emptyset$

// Означает, что символ « \emptyset » называется пустым множеством.

$\langle \text{переменная} \rangle ::= \langle \text{буква} \rangle \mid \langle \text{переменная} \rangle \langle \text{буква} \rangle \mid \langle \text{переменная} \rangle \langle \text{цифра} \rangle$

// Означает, что любая конечная последовательность букв и цифр,

// которая начинается с буквы, называется переменной.

$\langle \text{список множеств} \rangle ::= \langle \text{множество} \rangle \mid \langle \text{множество} \rangle, \langle \text{список множеств} \rangle$

²⁵ Данный раздел носит предварительный характер. Позже будут даны формальные определения основных понятий, введенных в этом разделе на полуформальном уровне.

// Означает, что любая непустая конечная последовательность множеств,
 // перечисленных через запятую, называется списком множеств.

$$\begin{aligned} \langle \text{множество} \rangle ::= & \langle \text{пустое множество} \rangle \mid \langle \text{переменная} \rangle \mid \\ & \mid \{ \langle \text{список множеств} \rangle \} \mid \\ & \mid 2^{\langle \text{множество} \rangle} \mid \langle \text{множество} \rangle^{\langle \text{множество} \rangle} \mid \\ & \mid \langle \langle \text{множество} \rangle \setminus \langle \text{множество} \rangle \rangle \mid \\ & \mid \langle \langle \text{множество} \rangle \cup \langle \text{множество} \rangle \rangle \mid \\ & \mid \langle \langle \text{множество} \rangle \cap \langle \text{множество} \rangle \rangle \mid \\ & \mid \langle \langle \text{множество} \rangle \times \langle \text{множество} \rangle \rangle \end{aligned}$$

// Означает, что множества получаются из пустого множества
 // и переменных для множеств в результате явного перечисления
 // в фигурных скобках «{...}»
 // и выполнения операций множества всех подмножеств,
 // множества всех функций из множества во множество,
 // разности, объединения (в двух формах), пересечения (в двух формах)
 // и (декартового) произведения (тоже в двух формах).

$$\begin{aligned} \langle \text{элементарное свойство} \rangle ::= & \langle \text{множество} \rangle \in \langle \text{множество} \rangle \mid \\ & \mid \langle \text{множество} \rangle \notin \langle \text{множество} \rangle \mid \\ & \mid \langle \text{множество} \rangle \subseteq \langle \text{множество} \rangle \mid \\ & \mid \langle \text{множество} \rangle \subset \langle \text{множество} \rangle \mid \\ & \mid \langle \text{множество} \rangle = \langle \text{множество} \rangle \mid \\ & \mid \langle \text{множество} \rangle \neq \langle \text{множество} \rangle \end{aligned}$$

// Означает, что элементарные свойства множеств – это различные
 высказывания о включении или равенстве множеств.

$$\begin{aligned} \langle \text{свойство} \rangle ::= & \langle \text{элементарное свойство} \rangle \mid \neg(\langle \text{свойство} \rangle) \mid \\ & \mid (\langle \text{свойство} \rangle \wedge \langle \text{свойство} \rangle) \mid (\langle \text{свойство} \rangle \vee \langle \text{свойство} \rangle) \mid \\ & \mid \exists \langle \text{переменная} \rangle (\langle \text{свойство} \rangle) \mid \forall \langle \text{переменная} \rangle (\langle \text{свойство} \rangle) \end{aligned}$$

// Означает, что свойства множеств,
 // которые можно использовать в аксиоме выделения,
 // конструируются из элементарных свойств при помощи
 // отрицания, конъюнкции и дизъюнкции,
 // кванторов существования и всеобщности.

Синтаксис языка математической логики состоит из формул. Формулы строятся из неопределяемых символов (которые в математической логике называются сигнатурой) при помощи отрицания, конъюнкции и дизъюнкции, кванторов существования и всеобщности. Можно сказать, что сигнатура – это алфавит языка. Обычно сигнатуру разбивают на три непересекающихся множества: переменные, символы отношений и символы операций. Каждый символ отношения и операции имеет специальную характеристику (атрибут) «местность», определяющую число аргументов, используемое с этим символом. Часто используются синонимы: символ предиката – вместо символа отношения, символ функции – вместо символа операции. Есть только два символа 0-местных отношений «TRUE» и «FALSE», их часто называют булевскими константами. Наоборот, 0-местных символов операций может быть сколько угодно, их обычно называют символами констант. Другие (не 0-местные) символы отношений и операций могут быть разделены на классы префиксных, инфиксных и постфиксных символов (в зависимости от того, где по отношению к аргументам принято записывать символ отношения или функции).

Терм в математической логике – это аналог понятия выражения в алгебре или программировании.

Определение 1. Термы получаются из переменных и символов операций следующим образом:

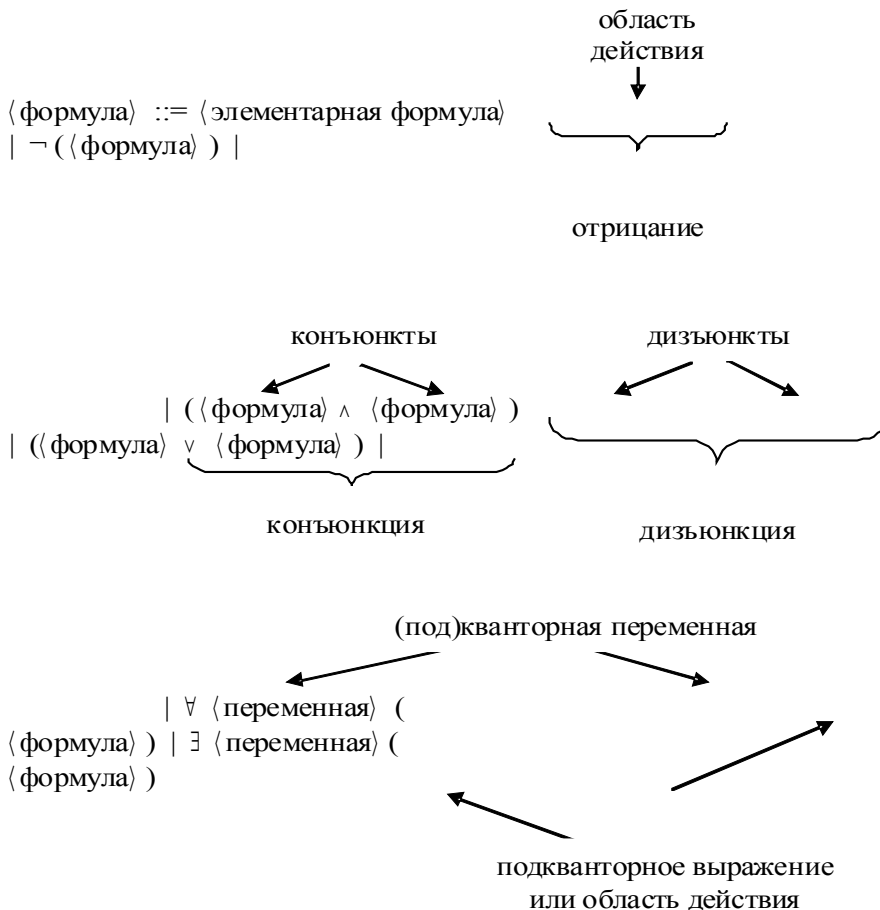
- всякая переменная является термом;
- всякий символ константы является термом;
- $f(t)$ является термом для всякого префиксного символа операции f и списка термов t , состоящего из термов, перечисленных через запятую, длина которого равна местности символа f , например, $\sin(x)$;
- (t) является термом для всякого инфиксного символа операции f и списка термов t , состоящего из термов, разделённых символом f , длина которого равна местности символа f , например, $(x + y)$;
- $(t)f$ является термом для всякого постфиксного символа операции f и списка термов t , состоящего из термов, перечисленных через запятую, длина которого равна местности символа f , например, $x++$.

Определение 2. Элементарная формула – это символ отношения со списком аргументов, а именно:

- TRUE и FALSE являются элементарными формулами;
- $r(t)$ является элементарной формулой для всякого префиксного символа отношения r и списка термов t , состоящего из термов, перечисленных через запятую, длина которого равна местности символа r ;

- t является элементарной формулой для всякого инфиксного символа отношения r и списка термов t , состоящего из термов, разделённых символом r , длина которого равна местности символа r ;
- $(t)r$ является элементарной формулой для всякого постфиксного символа отношения r и списка термов t , состоящего из термов, перечисленных через запятую, длина которого равна местности символа r .

Определение 3. Понятие формулы определим чисто синтаксическим путём с использованием нотации Бекуса–Наура:



Определение 4. Все вхождения кванторной переменной в область действия квантора называются связанным, а все остальные вхождения – свободными. Свободная переменная формулы – это переменная, не имеющая связанных вхождений в формулу. Замкнутая формула (или предложение или высказывание) – это формула без свободных переменных.

Определение 5. Модель M – это пара вида (D, I) , где $D \neq \emptyset$ – непустое множество, которое называется областью (или доменом) модели, а I – интерпретация символов отношений и операций подмножествами и функциями на D в соответствии с их местностью:

{

$I(f): D^n \rightarrow D$ для каждого символа n -местной операции f ,
 $I:$
 $I(r) \subseteq D^n$ для каждого символа n -местного отношения r .

Выберем и зафиксируем на время (пока не оговорим противное) произвольную модель $M = (D, I)$.

Определение 6. Состояние s (над M) – это произвольное отображение s , которое сопоставляет каждой переменной z некоторое значение $s(z) \in D$ из домена D . Множество всех состояний (над M) обозначим $SP(M)$. Интерпретация I позволяет доопределить значения любого состояния s на термах индукцией по их структуре: для переменных значения уже определены, а для любого символа операции f и любых термов t_1, \dots, t_n :

3. $s(f(t_1, \dots, t_n)) = I(f)(s(t_1), \dots, s(t_n))$, если f – префиксный символ,
4. $s(t_1 f \dots t_n) = I(f)(s(t_1), \dots, s(t_n))$, если f – инфиксный символ,
5. $s((t_1, \dots, t_n) f) = I(f)(s(t_1), \dots, s(t_n))$, если f – постфиксный символ.

Определение 7. Семантика формулы ϕ в модели $M = (D, I)$ – это множество состояний $M(\phi)$, где формула верна. Это множество определяется индукцией по структуре формул:

- $M(\text{TRUE}) = D$ и $M(\text{FALSE}) = \emptyset$;
- $M(r(t_1, \dots, t_n)) = \{s : (s(t_1), \dots, s(t_n)) \in I(r)\}$ для любого префиксного символа отношения и термов t_1, \dots, t_n ;
- $M(t_1 r \dots t_n) = \{s : (s(t_1), \dots, s(t_n)) \in I(r)\}$ для любого инфиксного символа отношения и термов t_1, \dots, t_n ;
- $M((t_1, \dots, t_n)r) = \{s : (s(t_1), \dots, s(t_n)) \in I(r)\}$ для любого постфиксного символа отношения и термов t_1, \dots, t_n ;
- $M(\neg(\psi)) = SP(M) \setminus M(\psi)$, $M(\psi \wedge \xi) = M(\psi) \cap M(\xi)$ и $M(\psi \vee \xi) = M(\psi) \cup M(\xi)$ для любых формул ψ и ξ ;
- $M(\forall z \psi) = \{s : s' \in M(\psi) \text{ для любого } s', \text{ которое может отличаться от } s \text{ только значением переменной } z\}$ для любой формулы ψ и переменной z ;
- $M(\exists z \psi) = \{s : s' \in M(\psi) \text{ для некоторого } s', \text{ которое может отличаться от } s \text{ только значением переменной } z\}$ для любой формулы ψ и переменной z .

Определение 8. Семантика позволяет определить 3-местное отношение выполнимости «состояние $\models_{\text{модель}}$ формула»: $s \models_M \phi \Leftrightarrow s \in M(\phi)$.

В соответствии с определением семантики в модели M и отношения выполнимости \models_M для любого состояния s имеют место следующие соотношения:

- $s \models_{\mathbf{M}} \text{TRUE}$, но неверно $s \models_{\mathbf{M}} \text{FALSE}$;
 - $s \models_{\mathbf{M}} (t_1 \dots t_n) \Leftrightarrow (s(t_1), \dots, s(t_n)) \in I(r)$ для префиксного символа отношения и термов t_1, \dots, t_n ;
 - $s \models_{\mathbf{M}} (t_1 \ r \dots t_n) \Leftrightarrow (s(t_1), \dots, s(t_n)) \in I(r)$ для инфиксного символа отношения и термов t_1, \dots, t_n ;
 - $s \models_{\mathbf{M}} (t_1 \dots t_n)r \Leftrightarrow (s(t_1), \dots, s(t_n)) \in I(p)$ для постфиксного символа отношения и термов t_1, \dots, t_n ;
 - $s \models_{\mathbf{M}} (\neg(\psi)) \Leftrightarrow$ неверно $s \models_{\mathbf{M}} \psi$; $s \models_{\mathbf{M}} (\psi \wedge \xi) \Leftrightarrow s \models_{\mathbf{M}} \psi$ и $s \models_{\mathbf{M}} \xi$; $s \models_{\mathbf{M}} (\psi \vee \xi) \Leftrightarrow s \models_{\mathbf{M}} \psi$ или $s \models_{\mathbf{M}} \xi$;
 $s \models_{\mathbf{M}} (\forall z \psi) \Leftrightarrow s' \models_{\mathbf{M}} \psi$ для любого s' , которое может отличаться от s только значением переменной z ;
 - $s \models_{\mathbf{M}} (\exists z \psi) \Leftrightarrow s' \models_{\mathbf{M}} \psi$ для некоторого s' , которое может отличаться от s только значением переменной z .
- Легко видеть, что $M(\varphi) = \{ s : s \models_{\mathbf{M}} \varphi \}$.

Классификация логик

Определение 9. Обозначим всё множество переменных V . Предположим, что это множество разбито на несколько непересекающихся подмножеств $\cup_{i \in I} V_i$, где I – или некоторый отрезок натуральных чисел вида $[1..K]$, или всё множество натуральных чисел \mathbf{N} . В таком случае можно рассматривать специальные модели $\mathbf{M} = (D, I)$, у которых домен D также может быть представлен в виде объединения непересекающихся множеств $\cup_{i \in I} D_i$, где $D_i \neq \emptyset$, а $D_{i+1} = 2^{D_i}$ для любого $i \in I$, $i < K$. Если рассматривать только такие состояния s над \mathbf{M} , что $s(z) \in D_i$ для любого $i \in I$ и любой переменной $z \in V_i$, то в таком случае говорят о логике K -ого порядка (когда $I = [1..K]$), и, соответственно, о логике высшего порядка (при $I = \mathbf{N}$).

Рассмотрим с точки зрения классификации логик по порядкам такие учебные предметы, как:

- элементарная алгебра, изучаемая в старших классах средней школы;
- математический анализ, изучаемый на первых курсах вузов;
- функциональный анализ, который изучается на 3 – 4 курсах на факультетах физико-математического профиля.

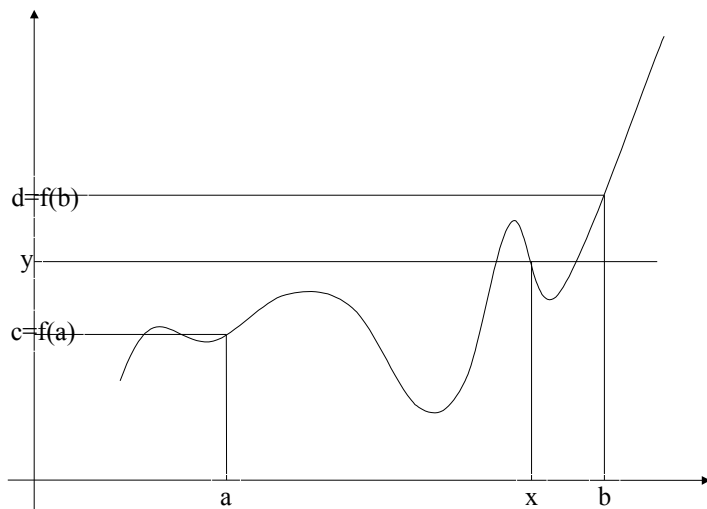
Элементарная алгебра занимается решением (систем) уравнений и неравенств в вещественных числах; эти уравнения заданы многочленами с параметрическими коэффициентами, которые обозначают опять же вещественные числа. Например, типичными задачами элементарной алгебры является нахождение всех вещественных корней квадратного уравнения с вещественными коэффициентами $a \cdot x^2 + b \cdot x + c = 0$, а также решение в вещественных

венных числах систем линейных неравенств с вещественными коэффициентами:

$$\begin{aligned} a_1 \cdot x + b_1 \cdot y &\geq c_1, \\ a_2 \cdot x + b_2 \cdot y &\leq c_2. \end{aligned}$$

В частности, в курсе элементарной алгебры доказывается, что для любых вещественных значений коэффициентов уравнение $a \cdot x^2 + b \cdot x + c = 0$ имеет решение в вещественных числах тогда и только тогда, когда $b^2 - 4 \cdot a \cdot c \geq 0$, т. е., что следующее предложение логики первого порядка является истинным в поле вещественных чисел: $\forall a \forall b \forall c (b^2 - 4 \cdot a \cdot c \geq 0 \leftrightarrow \exists x (a \cdot x^2 + b \cdot x + c = 0))$. Аналогично в курсе элементарной алгебры доказывается, что для любых вещественных значений a_1 и b_1 , a_2 и b_2 , c_1 и c_2 система неравенств $a_1 \cdot x + b_1 \cdot y \geq c_1$ и $a_2 \cdot x + b_2 \cdot y \leq c_2$ имеет решения в вещественных числах тогда и только тогда, когда $a_1 \cdot b_2 - a_2 \cdot b_1 \neq 0$ или когда $a_1 \cdot b_2 - a_2 \cdot b_1 = 0$, но при этом $c_1 \cdot b_2 \leq c_2 \cdot b_1$, т. е. что следующее предложение логики первого порядка является истинным в поле вещественных чисел: $\forall a_1 \forall b_1 \forall a_2 \forall b_2 \forall c_1 \forall c_2 ((a_1 \cdot b_2 \neq a_2 \cdot b_1 \vee a_1 \cdot b_2 = a_2 \cdot b_1 \wedge c_1 \cdot b_2 \leq c_2 \cdot b_1) \leftrightarrow \exists x \exists y (a_1 \cdot x + b_1 \cdot y \geq c_1 \wedge a_2 \cdot x + b_2 \cdot y \leq c_2))$. Таким образом, можно предположить, что курс элементарной алгебры формализуем в терминах логики первого порядка над полем вещественных чисел.

Математический анализ занимается изучением свойств непрерывных и дифференцируемых функций вещественной переменной. Примером может служить теорема о промежуточном значении непрерывной функции на замкнутом отрезке, авторство которой приписывается К. Т. В. Веерштрассу, Б. Больцано и О. Л. Коши: Для любой непрерывной функции f на отрезке вещественных чисел $[a..b]$, для любого числа $y \in [f(a)..f(b)]$ найдётся такое число $x \in [a..b]$, что $f(x) = y$.



Теперь вспомним, что всякая вещественная функция – это просто множество пар чисел. Поэтому приведённая теорема может быть сформулирована в терминах логики второго порядка над полем вещественных чисел следующим образом: $\forall f \in 2^{\mathbb{R} \times \mathbb{R}} \forall a \in \mathbb{R} \forall b \in \mathbb{R} \forall c \in \mathbb{R} \forall d \in \mathbb{R} \forall y \in \mathbb{R} (C(f) \wedge (a, c) \in f \wedge (b, d) \in f \wedge c \leq y \leq d \rightarrow \exists x \in \mathbb{R} (a \leq x \leq b \wedge (x, y) \in f))$, где C – одноместное отношение «быть непрерывной вещественной функцией», которое легко определяется на языке логики второго порядка над полем вещественных чисел как конъюнкция следующих трёх свойств:

- $\forall x \in \mathbb{R} \exists y \in \mathbb{R} ((x, y) \in f)$,
- $\forall x \in \mathbb{R} \forall y' \in \mathbb{R} \forall y'' \in \mathbb{R} (((x, y') \in f \wedge (x, y'') \in f) \rightarrow y' = y'')$,
- $\forall x' \in \mathbb{R} \forall y' \in \mathbb{R} \forall x'' \in \mathbb{R} \forall y'' \in \mathbb{R} \forall \varepsilon \in \mathbb{R} \exists \delta \in \mathbb{R} ((\varepsilon > 0 \wedge \delta > 0 \wedge (x', y') \in f \wedge (x'', y'') \in f \wedge |x' - x''| < \delta) \rightarrow |y' - y''| < \varepsilon)$.

Подобным образом курс математического анализа формализуем в терминах логики второго порядка над полем вещественных чисел.

Функциональный анализ изучает свойства функционалов, т. е. отображений, которые сопоставляют функциям некоторые числа или функции. Часто изучаются свойства линейных операторов, к которым относятся, например, дифференцирование и интегрирование вещественных функций. Следовательно, функциональный анализ изучает, в частности, функции от

вещественных функций, значениями которых являются снова вещественные функции; поскольку вещественные функции являются элементами множества $F=2^{R \times R}$, то операторы являются элементами множества $2^{F \times F}$. Поэтому часть функционального анализа, изучающая теорию линейных операторов на функциях вещественной переменной может быть формализована в терминах логики третьего порядка над полем вещественных чисел.

Кроме классификации логик по порядкам, существуют ещё несколько «классов» логик внутри первого порядка.

Определение 10. Исчисление²⁶/логика предикатов с равенством – это логика 1-го порядка, сигнатура которой содержит символ двуместного отношения (обычно используется « $=$ »), интерпретируемый в моделях как равенство (тождество). Логику 1-го порядка, в сигнатуре которой нет выделенного символа, интерпретируемого в моделях тождеством, называют просто исчислением/логикой предикатов (или исчислением предикатов без равенства). Чистое исчисление/логика предикатов – это логика 1-го порядка, сигнатура которой не содержит символов функций. Алгебра – это логика 1-го порядка, единственный символ отношения в сигнатуре которой – символ равенства, интерпретируемый в моделях тождеством.

В приведенной выше классификации логик не нашлось места для пропозициональной логики, или, как ещё её называют, пропозиционального исчисления. Дело в том, что это «граничный случай», так сказать логика нулевого порядка, который нуждается в специальном «доопределении».

Определение 11. Пропозициональная логика – это логика без переменных (т. е. множество переменных – пусто), все термы в этой логике – это так называемые замкнутые термы (т. е. термы без переменных, построенные только из символов констант и символов операций), а все элементарные формулы – это так называемые атомы (т. е. элементарные формулы, в которых все термы – замкнутые).

Из этого определения следует, что синтаксис пропозициональной логики можно определить так:

⟨пропозициональная формула⟩ :: ⟨замкнутый атом⟩ |
 | ¬ (⟨пропозициональная формула⟩) |
 | (⟨пропозициональная формула⟩ ∧ ⟨пропозициональная формула⟩) |
 | (⟨пропозициональная формула⟩ ∨ ⟨пропозициональная формула⟩)

Пусть $M=(D, I)$ – произвольная модель. Формально семантику пропозициональных формул надо определять на пространстве состояний $SP(M)$,

²⁶ Термин «исчисление», вообще говоря, имеет некоторый специальный смысловой оттенок, к обсуждению которого мы вернёмся в лекции 4, когда будем определять понятие аксиоматической системы.

которое есть множество всех всюду определённых функций из множества переменных V в D . Но для пропозициональной логики множество переменных есть пустое множество \emptyset . Поэтому пространство состояний $SP(M)$ состоит из единственного состояния – пустой «функции» \emptyset , которая сопоставляет всякой переменной из пустого множества некоторое значение из D . Таким образом, $SP(M) = \{\emptyset\}$. Следовательно, для каждой пропозициональной формулы ϕ или $M(\phi) = \emptyset$, или $M(\phi) = \{\emptyset\}$. Какой из этих случаев имеет место можно определить в соответствии с общими правилами для семантики логических формул. Однако, традиционно семантику пропозициональной логики определяют с использованием понятия означивания.

Определение 11. Означивание E – это отображение, которое сопоставляет каждому атому ϕ значение $E(\phi)=И$ или $E(\phi)=Л$. Означивание E может быть распространено на множество всех формул посредством таблиц истинности:

ϕ	$\neg(\phi)$
И	Л
Л	И

ϕ	И	Л	И	Л
ψ	И	И	Л	Л
$(\phi \wedge \psi)$	И	Л	Л	Л

ϕ	И	Л	И	Л
ψ	И	И	Л	Л
$(\phi \vee \psi)$	И	И	И	Л

Утверждение 1. Пусть $M=(D, I)$ – произвольная модель, а E – такое означивание, что для любого атома ψ имеем:

$$E(\psi) = \begin{cases} И, & \text{если } M(\psi) \neq \emptyset \\ Л & \text{в противном случае.} \end{cases}$$

Тогда для любой пропозициональной формулы ϕ имеем: $E(\phi) = И \Leftrightarrow M(\phi) = \{\emptyset\}$ и $E(\phi) = Л \Leftrightarrow M(\phi) = \emptyset$.

Доказательство. Пропозициональные формулы построены из атомов при помощи связок для отрицания, конъюнкции и дизъюнкции. Поэтому доказательство проведём индукцией по числу отрицания, конъюнкции и дизъюнкции в пропозициональной формуле ϕ .

База индукции: φ не содержит связок, т. е. является атомом. Но в таком случае утверждение для φ прямо следует из определения означивания E .

Предположение индукции: пусть утверждение верно для всех пропозициональных формул, содержащих не более $k \geq 0$ связок.

Шаг индукции: разберём три возможных случая²⁷ $\varphi \equiv (\neg\theta)$, $\varphi \equiv (\theta \vee \chi)$ и $\varphi \equiv (\theta \wedge \chi)$.

- $\varphi \equiv (\neg\theta)$:
 1. $E(\varphi)=И \Leftrightarrow E(\theta)=Л \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\emptyset \Leftrightarrow M(\varphi)=SP(M) \setminus M(\theta) = \{\emptyset\};$
 2. $E(\varphi)=Л \Leftrightarrow E(\theta)=И \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\{\emptyset\} \Leftrightarrow M(\varphi)=SP(M) \setminus M(\theta) = \emptyset;$
- $\varphi \equiv (\theta \vee \chi)$:
 1. $E(\varphi)=И \Leftrightarrow E(\theta)=И \text{ или } E(\chi)=И \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\{\emptyset\} \text{ или } M(\chi)=\{\emptyset\} \Leftrightarrow M(\varphi)=M(\theta) \cup M(\chi) = \{\emptyset\};$
 2. $E(\varphi)=Л \Leftrightarrow E(\theta)=Л \text{ и } E(\chi)=Л \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\emptyset \text{ и } M(\chi)=\emptyset \Leftrightarrow M(\varphi)=M(\theta) \cup M(\chi) = \emptyset;$
- $\varphi \equiv (\theta \wedge \chi)$:
 1. $E(\varphi)=И \Leftrightarrow E(\theta)=И \text{ и } E(\chi)=И \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\{\emptyset\} \text{ и } M(\chi)=\{\emptyset\} \Leftrightarrow M(\varphi)=M(\theta) \cap M(\chi) = \{\emptyset\};$
 2. $E(\varphi)=Л \Leftrightarrow E(\theta)=Л \text{ или } E(\chi)=Л \Leftarrow (\text{по предположению индукции}) \Rightarrow M(\theta)=\emptyset \text{ или } M(\chi)=\emptyset \Leftrightarrow M(\varphi)=M(\theta) \cap M(\chi) = \emptyset.$

■

Лекция 4. Воспоминания о теории множеств и математической логике - II

Аксиоматические системы

При определении теоретико-множественной семантики мы определили 3-местное отношение выполнимости «состояние $\models_{\text{модель}}$ формула», позволяющее определить несколько «производных» отношений: истинности, тождественной истинности и реализуемости.

²⁷ Для формального языка знак « \Leftrightarrow » означает «синтаксически совпадает».

Определение 1.

- Двухместное отношение истинности « $\models_{\text{модель}}$ формула»: $\models_{\text{м}} \Phi \Leftrightarrow s \models_{\text{м}} \Phi$ для всякого состояния $s \in SP(M)$.
- Одноместное отношение тождественной истинности (или тавтологии) « \models формула»: $\models \Phi \Leftrightarrow \models_{\text{м}} \Phi$ для всякой модели M .
- Одноместное отношение реализуемости « $\models_{\text{—}}$ формула»: $\models_{\text{—}} \Phi \Leftrightarrow s \models_{\text{м}} \Phi$ для некоторых модели M и некоторого состояния $s \in SP(M)$.

Легло заметить, что формула является тавтологией тогда и только тогда, когда её отрицание не реализуется.

Можно также заметить, что за счёт «кванторов» по моделям («для всякой модели» и «для некоторых моделей») определения тавтологии и реализуемости имеют порядок больший, чем логика, для которой определяются эти понятия, так как кванторы варьируют интерпретации символов отношений и операций. Поэтому возникает естественное желание чисто синтаксически получать (вычислять или исчислять) тавтологии и реализуемые формулы, без сложных теоретико-множественных семантических рассуждений о моделях и состояниях. Классический формализм для решения этой проблемы – аксиоматические системы или исчисления²⁸.

Ключевое понятие в аксиоматизации – понятие правила вывода.

Определение 2. Каждое правило вывода имеет вид

⟨совокупность посылок⟩

⟨совокупность заключений⟩

где совокупности посылок и заключений – это некоторые «агрегаты», построенные из формул, множеств формул, списков формул и т. д. Аксиома –

²⁸ Идея (или мечта) заменить сложные смысловые (семантические) рассуждения и/или доказательства простыми синтаксическими выкладками в соответствии с набором простых правил для «вычисления» истинных утверждений владела умами многих учёных на протяжении, по крайней мере, четырёх веков (XVII–XX вв.). Именно с такой целью в XVII в. И. Ньютон и Г. В. фон Лейбниц разработали «исчисление» флюидов (бесконечно малых), в XVIII Л. Эйлер развивал дифференциальное, интегральное и вариационное «исчисления», в XIX в. Дж. Буль закладывал основы «исчисления» высказываний, а в XX в. Г. Фреге и Б. Рассел формализовали «исчисление» предикатов. Поэтому на протяжении XVII–XX вв. в философии и логике сложилась традиция «исчислением» в узком смысле этого слова называть именно синтаксические системы для «вычисления» истинны. Однако также сложилась математическая традиция расширительного использования термина «исчисление»: ни у кого не вызывает возражений названия «дифференциальное исчисление», «интегральное исчисление», «вариационное исчисление», поэтому теперь в логике также допускается расширительное использование понятия «исчисление» для разного рода «логик»; эта практика допустима в частности тогда, когда явно ставится цель построить синтаксическое исчисление для этих логик. Именно так обстоит дело с «исчислением предикатов» и «пропозициональным исчислением».

это правило вывода с пустым множеством посылок. Противоречие – это правило вывода с пустым множеством заключений. Аксиоматическая система – любое множество правил вывода.

В качестве примера аксиоматической системы рассмотрим следующую аксиоматизацию АХ пропозициональной логики. Совокупности посылок и заключений в этой системе служат конечные последовательности множеств формул пропозициональной логики. Сама системы АХ – это бесконечное множество, содержащее правила вывода следующего вида:

$$\begin{array}{c}
 \frac{\{\varphi, \neg(\varphi)\} \cup \Sigma}{\{\varphi\} \cup \Sigma} \\
 \hline
 \frac{}{\{\neg(\neg(\varphi))\} \cup \Sigma} \\
 \hline
 \frac{\{(\neg(\varphi) \vee \neg(\psi))\} \cup \Sigma}{\{(\neg(\varphi \wedge \psi))\} \cup \Sigma} \\
 \hline
 \frac{\{(\neg(\varphi) \wedge \neg(\psi))\} \cup \Sigma}{\{\neg(\varphi \vee \psi)\} \cup \Sigma} \\
 \hline
 \frac{\{\varphi, \psi\} \cup \Sigma}{\{(\varphi \vee \psi)\} \cup \Sigma} \\
 \hline
 \frac{\{\varphi\} \cup \Sigma \quad \{\psi\} \cup \Sigma}{\{(\varphi \wedge \psi)\} \cup \Sigma}
 \end{array}$$

(Здесь φ и ψ – пропозициональные формулы, а Σ – произвольное конечное множество пропозициональных формул.)

В этой аксиоматической системе все совокупности заключений представляют собой одиночное множество пропозициональных формул, а совокупности посылок – пустую (первая аксиома), одноэлементную (все прави-

ла, кроме первого и последнего) или двухэлементную (последнее правило) последовательность множеств пропозициональных формул.

В записи аксиоматических систем часто используют схемы правила вывода.

Определение 3. Схема правила вывода имеет вид

$$\frac{\langle \text{совокупность схем посылок} \rangle}{\langle \text{совокупность схем заключений} \rangle}, \langle \text{синтаксические ограничения} \rangle,$$

где схемы посылок и заключений используют метапеременные для формул, множеств формул и т. п., вместо которых можно подставлять любые формулы, множества формул и т. д. Правила вывода тогда получаются из схем после подстановки вместо всех метапеременных конкретных формул, множеств формул и т. е. Понятие схемы аксиомы – это частный случай понятия схемы правила вывода.

В качестве примера задания аксиоматической системы конечным набором схем правил вывода рассмотрим ту же аксиоматизацию АХ пропозициональной логики. Теперь, однако, в силу сложившейся традиции фигурные скобки «{...}» вокруг множеств формул будут опущены, а вместо значка объединения « \cup » будет использоваться запятая « $,$ ». В записи схем правил вывода используют ϕ, ψ и т. д. в качестве метапеременных для пропозициональных формул, а Σ и т. д. – в качестве метапеременной для конечных множеств пропозициональных формул.

$$\frac{\phi, \psi}{\phi, \psi} \quad \frac{\phi, \psi}{\phi, \psi} \quad \frac{\phi, \psi}{\phi, \psi} \quad \frac{\phi, \psi}{\phi, \psi}$$

Определение 4. Вывод (или дерево вывода) в аксиоматической системе – это дерево конечной высоты, узлы которого помечены «агрегатами» из формул логики так, что для любого узла выполняется следующее: если узел

помечен каким-либо «агрегатом» agr , а все его непосредственные наследники помечены какими-либо «агрегатами» $\text{agr}_1, \dots, \text{agr}_n$, то

$$\frac{\text{agr}_1 \dots \text{agr}_n}{\text{agr}}$$

является правилом вывода данной аксиоматической системы. Доказательство в аксиоматической системе – это дерево вывода, все листья которого помечены заключениями аксиом. «Агрегат» из формул называется доказуемым в аксиоматической системе, если существует доказательство, корень которого помечен этим агрегатом.

Альтернативой древовидному выводу и доказательству является линейный вывод и доказательство.

Определение 5. Пусть дано некоторое дерево вывода в аксиоматической системе. Линейное представление этого дерева вывод (или просто линейный вывод) – конечная последовательность нумерованных строк, каждая из которых соответствует единственному узлу дерева и содержит «агрегат» из формул, соответствующий этому узлу, и список номеров членов этой последовательности, которые соответствуют наследникам узла, причем, если m -й член этой последовательности ссылается на номера m_1, \dots, m_n , то $m > m_1, \dots, m_n$. В случае, если линейно представлено дерево доказательства, говорят о линейном доказательстве.

Очевидно также, что линейный вывод и доказательство является менее наглядным для человека, но более компактным и более пригодным для обработки на компьютере, поэтому пока мы не будем использовать линейное представление для вывода и доказательств в аксиоматических системах.

В том случае, когда с заключениями правил вывода можно связать некоторым способом формулы некоторой логики, можно определить понятия совместимости и надёжности аксиоматической системы.

Определение 6. Пусть SA – аксиоматическая система, LG – логика, а F – отображение, которое сопоставляет каждому заключению всякого правила вывода из SA некоторую формулу логики LG . Для любого заключения CN будем писать $\vdash_{SA} F(CN)$ и говорить что формула $F(CN)$ доказуема в SA , когда CN доказуемо в этой аксиоматической системе. Аксиоматическая система называется

- совместимой, когда для любого заключения CN доказуемость $\vdash_{SA} F(CN)$ влечёт реализуемость $_ \vdash F(CN)$;
- надёжной, когда для любого заключения CN доказуемость $\vdash_{SA} F(CN)$ влечёт тождественную истинность $\models F(CN)$.

В том случае, когда с формулами некоторой логики можно связать некоторым способом заключения правил вывода, можно определить понятие полноты аксиоматической системы.

Определение 7. Пусть SA – аксиоматическая система, LG – логика, а G – отображение, которое сопоставляет каждой формуле логики LG заключение некоторого правила вывода из SA . В таком случае аксиоматическая система называется полной, если для любой формулы φ этой логики тождественная истинность $(\models \varphi)$ влечёт доказуемость $G(\varphi)$ в SA .

Неформально говоря, совместимость означает, что заведомой бессмысленности, ложного заключения нельзя доказать, надёжность означает, что всё доказуемое – заведомо истинно, а полнота означает, что всё истинное – доказуемо.

Аксиоматизация пропозициональной логики

Заключения всех правил вывода аксиоматической системы AX – конечные множества пропозициональных формул. Поэтому отображение $F = (\lambda \Sigma. (\bigvee_{\varphi \in \Sigma} \varphi))$ сопоставляет каждому заключению Σ в AX пропозициональную формулу $F(\Sigma) = (\bigvee_{\varphi \in \Sigma} \varphi)$, а соответствие $G = (\lambda \psi. \{\psi\})$ каждой пропозициональной формуле ψ сопоставляет заключение (состоящее из единственной формулы) $G(\psi) = \{\psi\}$ в AX .

Утверждение 1. Аксиоматическая система AX для пропозициональной логики совместна и надёжна (при соответствии $(F = \lambda \Sigma. (\bigvee_{\varphi \in \Sigma} \varphi))$), а также полна (при соответствии $G = (\lambda \psi. \{\psi\})$).

Доказательство. Заметим, что для любых формул пропозициональной логики φ и ψ , для любого конечного множества пропозициональных формул Σ следующие формулы являются тавтологиями²⁹:

$$\left(\begin{array}{l} (\varphi \vee \neg(\varphi)) \vee (\vee \Sigma), \\ (\neg(\neg(\varphi)) \vee (\vee \Sigma)) \leftrightarrow (\varphi \vee (\vee \Sigma)), \\ (\neg(\varphi \wedge \psi) \vee (\vee \Sigma)) \leftrightarrow ((\neg(\varphi) \vee \neg(\psi)) \vee (\vee \Sigma)), \\ (\neg(\varphi \vee \psi) \vee (\vee \Sigma)) \leftrightarrow ((\neg(\varphi) \wedge \neg(\psi)) \vee (\vee \Sigma)), \\ ((\varphi \vee \psi) \vee (\vee \Sigma)) \leftrightarrow (\varphi \vee (\psi \vee (\vee \Sigma))), \\ ((\varphi \wedge \psi) \vee (\vee \Sigma)) \leftrightarrow ((\varphi \vee (\vee \Sigma)) \wedge (\psi \vee (\vee \Sigma))). \end{array} \right) \quad (\text{Prop_Taut})$$

Так как любое правило вывода из AX представимо в виде

²⁹ Для проверки достаточно составить таблицы истинности этих формул.

$$\frac{\Sigma_i, i \in I}{\Sigma},$$

где Σ – множество-заключение, все Σ_i – множества-посылки, индексированные $i \in I$, а множество индексов I – это \emptyset для аксиом, $\{1\}$ для однопосылочных правил или $\{1, 2\}$ для двухпосылочных правил, то в силу замеченных тавтологий (Prop_Taut) следующая формула тоже является тавтологией:

$$(\bigvee_{\varphi \in \Sigma} \varphi) \leftrightarrow (\bigwedge_{i \in I} (\bigvee_{\psi \in \Sigma_i} \psi))$$

(где конъюнкция по пустому множеству индексов обозначает формулу TRUE). В терминах соответствия F последняя тавтология может быть переписана в виде $F(\Sigma) \leftrightarrow \bigwedge_{i \in I} F(\Sigma_i)$ для всех правил вывода в AX ; для аксиом, в частности, эта тавтология переписывается в виде $F(\Sigma) \leftrightarrow \text{TRUE}$, что означает, что $F(\Sigma)$ является тавтологией.

Следовательно,

- для любой аксиомы в AX
$$\frac{}{\Sigma}$$

 $F(\Sigma)$ является тавтологией;
- для любого правила вывода в AX

$$\frac{\Sigma_i, i \in I}{\Sigma}$$

$F(\Sigma)$ является тавтологией тогда и только тогда, когда все $F(\Sigma_i)$, $i \in I$, являются тавтологиями.

Поэтому для любого узла Σ в любом дереве доказательства $F(\Sigma)$ является тавтологией. Таким образом, аксиоматическая система AX для пропозициональной логики совместна и надежна.

Полнота аксиоматической системы AX для пропозициональной логики следует из вышеприведённых фактов и трёх дополнительных наблюдений. Во-первых, $F(G(\psi)) = \psi$ для любой пропозициональной формулы ψ . Во-вторых, для любого множества пропозициональных формул Σ , в котором есть хотя бы один из знаков « \wedge » или « \vee », в AX есть правило вывода, в котором Σ является заключением. И в-третьих, для любого правила вывода

$$\Sigma_i, i \in I$$

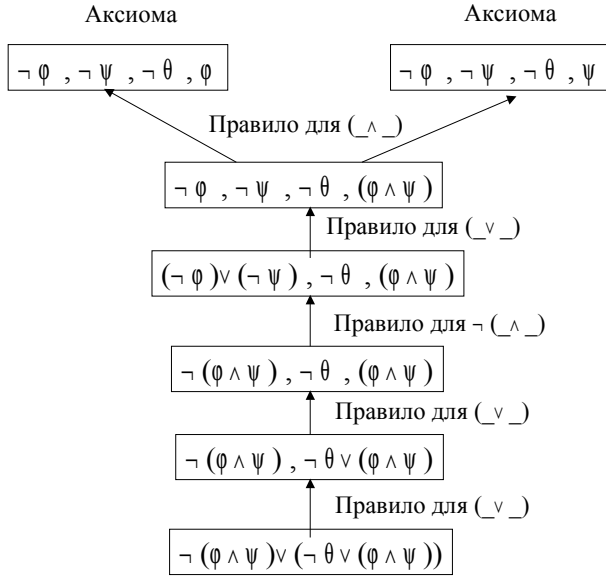
Σ

из АХ каждое множество-посылка $\Sigma_i, i \in I$ «проще», чем множество-закл^ючение Σ : или в посылка меньше знаков « \wedge » и « \vee », или знак « \neg » применяется к более простым формулам.

Для доказательства какой-либо пропозициональной тавтологии ψ достаточно попытаться построить дерево доказательства в обратном порядке, начиная с $\{\psi\}$ и применяя любые правила вывода. В результате получится некоторое дерево РТ, которое, возможно, бесконечно и (возможно) содержит листья, не являющиеся аксиомами. Но в силу того, что каждый раз посылки будут проще заключений, все пути в этом дереве конечны, и, следовательно, всё дерево РТ конечно (так как имеет ветвление не более 2). В то же время, поскольку корень этого дерева $\{\psi\}$, причём ψ – тавтология, то для всякого узла Σ в этом дереве $F(\Sigma)$ – тоже тавтология в силу доказательства надёжности АХ. В соответствии с построением этого дерева в нем для любого листа неприменимо ни одно из правил вывода с непустой посылкой. Следовательно, любой лист Σ в РТ не содержит \wedge и \vee , представляет собой совокупность атомов и отрицаний атомов, при этом $F(\Sigma)$ является тавтологией. В соответствии с таблицами истинности для любой дизъюнкции атомов и отрицаний атомов $(\vee_{j \in J} \phi_j)$ имеем: $(\vee_{j \in J} \phi_j)$ является тавтологией тогда и только тогда, когда $(\vee_{j \in J} \phi_j)$ содержит некоторый атом и его отрицание. Поэтому для всякого листа Σ в построенном дереве РТ имеем: $F(\Sigma)$ является тавтологией тогда и только тогда, когда Σ содержит некоторый атом и его отрицание, т. е. является аксиомой. Поэтому для произвольной пропозициональной формулы ψ имеем: ψ является тавтологией $\Leftrightarrow F(G(\psi))$ является тавтологией $\Rightarrow G(\psi)$ имеет дерево доказательства в АХ. Таким образом, аксиоматическая система АХ полна. ■

Разберём пару примеров на доказательство пропозициональных формул в аксиоматической системе АХ. Сначала попробуем доказать тавтологию $(\phi \wedge \psi) \rightarrow (\theta \rightarrow (\psi \wedge \phi))$, а затем – покажем недоказуемость $\phi \rightarrow (\phi \wedge \psi)$, где импликация « $\xi \rightarrow \chi$ » служит сокращением, «макросом» для формулы $((\neg \xi) \vee \chi)$.

После подстановки $(\neg_ \wedge _)$ вместо $(_ \rightarrow _)$ первая формула приобретает вид $\neg(\phi \wedge \psi) \vee (\neg\theta \vee (\phi \wedge \psi))$. Для этой формулы возможно следующее доказательство:



Заметим, однако, что общепринятым является другой формат представления дерева доказательства, а именно:



Теперь рассмотрим вторую формулу $\varphi \rightarrow (\varphi \wedge \psi)$. После подстановки $(\neg _ \wedge _)$ вместо $(_ \rightarrow _)$ она приобретает вид $(\neg \varphi) \vee (\varphi \wedge \psi)$. Попробуем для неё построить дерево доказательства. Заметим, что на первом шаге безальтернативно мы должны применять только правило для $(_ \vee _)$. Далее возможно применение правила для $(_ \wedge _)$ или правил, продиктованных внутренней структурой формулы φ . Однако без дополнительной информации о структуре φ дерево можно построить только в следующем виде:

$$\begin{array}{c}
 \begin{array}{cc}
 \text{Аксиома} & ? \\
 \hline
 \neg \varphi, \varphi & \neg \varphi, \psi \\
 \hline
 \end{array} \\
 \hline
 (\neg \varphi), (\varphi \wedge \psi) \\
 \hline
 (\neg \varphi) \vee (\varphi \wedge \psi)
 \end{array}$$

Разумеется, построенное дерево не является деревом доказательства, но оно может быть достроено до полного дерева доказательства в том случае, когда ψ является тавтологией.

Система вывода для логик первого порядка

В этом пункте лекции будет рассказано, как в принципе можно «дополнить» систему АХ до непротиворечивой, надежной и полной системы для логики первого порядка. Однако надо иметь в виду, что предлагаемая здесь аксиоматическая система не является удобной для практических целей.

Рассмотрим совокупность $АХ^-$ дополнительные правила вывода для логики первого порядка, которые получаются в результате «обращения» (перевёртывания) правил вывода (кроме аксиом) системы АХ:

$$\begin{array}{c}
\frac{\neg(\neg(\varphi)), \Sigma}{\varphi, \Sigma} \\
\\
\frac{\neg(\varphi \vee \psi), \Sigma}{(\neg(\varphi) \wedge \neg(\psi)), \Sigma} \qquad \frac{\neg(\varphi \wedge \psi), \Sigma}{(\neg(\varphi) \vee \neg(\psi)), \Sigma} \\
\\
\frac{(\varphi \vee \psi), \Sigma}{\varphi, \psi, \Sigma} \\
\\
\frac{(\varphi \wedge \psi), \Sigma}{\varphi, \Sigma} \qquad \frac{(\varphi \wedge \psi), \Sigma}{\psi, \Sigma}
\end{array}$$

Далее обозначим через QX систему из правил вывода для логики первого порядка, заданных следующими шестью схемами:

$$\begin{array}{c}
\frac{\exists x(\neg(\varphi)), \Sigma}{\neg(\forall x(\varphi)), \Sigma} \qquad \frac{\forall x(\neg(\varphi)), \Sigma}{\neg(\exists x(\varphi)), \Sigma} \\
\\
\frac{\bigvee_{t \in T} \varphi(t), \Sigma}{\exists x(\varphi(x)), \Sigma} \quad \text{для любого конечного множества термов } T \\
\\
\frac{\varphi(x), \Sigma}{\forall x(\varphi(x)), \Sigma} \quad \text{для любой переменной } x, \text{ которая не входит свободно в } \Sigma \\
\\
\frac{\forall x(\varphi(x)), \Sigma}{\varphi(t), \Sigma} \quad \text{для любого терма } t \\
\\
\frac{\varphi, \Sigma \quad \neg(\varphi), \Sigma}{\Sigma} .
\end{array}$$

Обозначим через BX аксиоматическую систему $(AX \cup AX^- \cup QX)$.

Для любого множества формул первого порядка Φ обозначим через $M(\Phi)$ совокупность всех моделей, в которых истинны все формулы из множества Φ , а через $BX(\Phi)$ – аксиоматическую систему BX , дополненную в качестве аксиом формулами Φ , т. е. аксиомами заданными схемой

$$\frac{}{\varphi, \Sigma},$$

где $\varphi \in \Phi$, а Σ – произвольное конечное множество формул первого порядка.

Утверждение 2. Для любого множества Φ формул первого порядка аксиоматическая система $BX(\Phi)$ надежна для логики первого порядка в классе моделей $M(\Phi)$ в следующем смысле: для всякого множества формул Σ , если $\vdash_{BX(\Phi)} (\vee \Sigma)$, то $\vdash_M (\vee \Sigma)$ для всякой модели $M \in M(\Phi)$.

Оставим это утверждение без доказательства, но отметим, что его можно провести индукцией по высоте дерева доказательства Σ .

Если опять (как и в пропозициональном случае) принять $F = (\lambda \Sigma. (\vee_{\varphi \in \Sigma} \varphi))$, то из утверждений 2 (при $\Phi = \emptyset$) следует, что аксиоматическая система BX совместна и надежна для логики первого порядка. Действительно, если $\Phi = \emptyset$, то любая формула $\varphi \in \Phi$ истинна в любой модели. Следовательно, при $\Phi = \emptyset$ имеем $M(\Phi)$ – это все модели вообще. Поэтому для любого множества формул первого порядка Σ имеем: если $\vdash_{BX(\emptyset)} (\vee \Sigma)$, то $\vdash_M (\vee \Sigma)$ для любой модели M .

Эбрановы модели непротиворечивых теорий³⁰

Напомним, что предложение – это любая замкнутая формула.

Определение 8. Пусть L – некоторая логика. Теория (в L) – это любое множество предложений (логики L). Полная теория (в L) – это такая теория Θ , что для любого предложения φ логики L или само это предложение φ , или его отрицание $\neg(\varphi)$ содержатся в этой теории Θ . Будем говорить, что теория первого порядка Θ непротиворечива, если в аксиоматической системе $BX(\Theta)$ недоказуема тождественно ложное предложение.

Оставим без доказательства следующие два утверждения.

Утверждение 3. Всякая непротиворечивая теория первого порядка может быть расширена до полной непротиворечивой теории в той же логике первого порядка, что и исходная теория.

Утверждение 4. Пусть L – произвольная логика первого порядка, а Θ – произвольная непротиворечивая теория в этой логике. Пусть c_1, \dots, c_n, \dots – счетный список новых символов констант, а $\varphi_1(z), \dots, \varphi_n(z), \dots$ произвольный

³⁰ Данный пункт носит факультативный характер и может быть опущен при чтении.

пересчёт всех формул с (не более чем одной) свободной переменной z логики L , обогащенной этими символами, такой, что для любого $n \geq 0$ символ c_n не встречается в формулах $\varphi_1(z), \dots, \varphi_n(z)$. Пусть теория Θ_0 есть Θ . Для любого $n \geq 0$ пусть теория Θ_{n+1} есть $\Theta_n \cup \{\exists z (\varphi_{n+1}(z)) \rightarrow \varphi_{n+1}(c_{n+1})\}$. Тогда $(\bigcup_{n \geq 0} \Theta_n)$ - непротиворечивая теория логики L , обогащенной символами c_1, \dots, c_n, \dots

Утверждение 5. Всякая непротиворечивая теория первого порядка имеет счётную модель.

Доказательство (набросок). Выберем произвольно логику первого порядка L и непротиворечивую теорию Θ в этой логике. Пусть c_1, \dots, c_n, \dots – счетный список новых символов констант, а $\varphi_1(z), \dots, \varphi_n(z), \dots$ произвольный пересчет всех формул с (не более чем одной) свободной переменной z логики L , обогащенной этими символами, такой, что для любого $n \geq 0$ символ c_n не встречается в формулах $\varphi_1(z), \dots, \varphi_n(z)$. Пусть теория Θ_0 есть Θ . Для любого $n \geq 0$ пусть теория Θ_{n+1} есть $\Theta_n \cup \{\exists z (\varphi_{n+1}(z)) \rightarrow \varphi_{n+1}(c_{n+1})\}$. Согласно утверждению 4, $(\bigcup_{n \geq 0} \Theta_n)$ – непротиворечивая теория логики L , обогащенная символами c_1, \dots, c_n, \dots . В соответствии с утверждением 3 эта теория может быть расширена до некоторой полной непротиворечивой теории Ξ в этой же логике. Так называемый Эрбранов универсум для неё – это множество всех замкнутых термов, на котором всякий функциональный символ интерпретирован синтаксически, например: функция, соответствующая двуместному функциональному символу f , преобразует любую пару замкнутых термов (t_1, t_2) в замкнутый терм $f(t_1, t_2)$. Эрбранова модель $N(\Xi)$ для теории Ξ определяет интерпретацию предикатных символов на Эрбрановом универсуме через принадлежность теории Ξ , например: отношение, соответствующее одноместному предикатному символу q , состоит из тех и только тех замкнутых термов t , для которых $q(t) \in \Xi$. Индукцией по структуре предложений, на основе непротиворечивости Ξ , её полноты и строения (а именно: Ξ содержит все предложения $\exists z (\varphi_{n+1}(z)) \rightarrow \varphi_{n+1}(c_{n+1})$) можно показать, что для любого предложения φ логики L , обогащенной символами c_1, \dots, c_n, \dots , имеет место следующая эквивалентность: $\vdash_{\text{BX}(\Xi)} \varphi \Leftrightarrow \models_{N(\Xi)} \varphi$.

Если вам оказалось недостаточно этого конспективного доказательства утверждения 5 или вы хотите подробнее «вспомнить» теории первого порядка, то рекомендуем книгу Э. Мендельсона Введение в математическую логику (М.: Наука, 1976. Гл. 2. Теории первого порядка). Хотя приведенная в данной лекции аксиоматическая система BX и отличается от системы, принятой в книге Э. Мендельсона, основные идеи и конструкции повторя-

ют идеи и конструкции Э. Мендельсона. С другим подходом к теориям первого порядка можно познакомиться в Ю. Л. Ершова и Е. А. Палютина Математическая логика. (М., Наука, 1987. Гл. 3. Истинность на алгебраических системах; гл. 4. Исчисление предикатов.).

Классические теоремы для логик первого порядка

Утверждение 6. Для любой теории первого порядка Θ аксиоматическая система $BX(\Theta)$ полна для логики первого порядка в классе моделей $\mathbf{M}(\Theta)$ в следующем смысле: для всякого множества предложений Σ , если $\vdash_M (\vee \Sigma)$ для всякой модели $M \in \mathbf{M}(\Theta)$, то $\vdash_{BX(\Theta)} (\vee \Sigma)$.

Доказательство. Достаточно ограничиться только непротиворечивыми теориями. Для любой такой теории Θ и любого предложения φ , если неверно $\vdash_{BX(\Theta)} \varphi$, то теория $\Theta \cup \{\neg(\varphi)\}$ непротиворечива и, следовательно (по утверждению 5), имеет модель $M \in \mathbf{M}(\Theta)$, такую, что $\vdash_M \neg(\varphi)$. Поэтому если для некоторого множества предложений Σ имеет место $\vdash_M (\vee \Sigma)$ в любой модели $M \in \mathbf{M}(\Theta)$, но при этом неверно, что $\vdash_{BX(\Theta)} (\vee \Sigma)$, то, приняв $(\vee \Sigma)$ в качестве φ , получаем такую модель $M \in \mathbf{M}(\Theta)$, что $\vdash_M \neg(\varphi)$. Противоречие. ■

Аналог этого утверждения для аксиоматизации (отличной от BX) логики первого порядка впервые был доказан К. Гёделем в начале 1930-х гг. и с тех пор носит название теоремы полноты Гёделя. Сложилась традиция также называть все аналоги исходной теоремы полноты Гёделя для логик первого порядка.

Определение 9. Будем говорить, что множество формул реализуется, если существует модель и состояние, в которых реализуются все формулы этого множества. Множество формул называется

- локально совместным, если любое его конечное подмножество реализуется;
- глобально совместным, если само множество реализуется.

Утверждение 7.

- Если теория первого порядка совместна, то она имеет счетную модель.
- Всякая локально совместная теория первого порядка глобально совместна.

Доказательство. Достаточно заметить, что всякая глобально/локально совместная теория является непротиворечивой, а поэтому в соответствии с доказательством утверждения 5 имеет счётную модель – Эрбранов универсум. ■

Первую часть утверждения 7 независимо друг от друга доказали Л. Лёвенгейм и Т. Сколем в 1910-1920-х годах, а вторую – А. И. Мальцев в конце 1930-х гг. В нашем изложении обе части являются следствием утверждения 5, поэтому первая часть этого утверждения носит название теоремы Лёвенгейма-Сколема, а вторая – теоремы компактности Мальцева.

Эти классические теоремы имеют два на первый взгляд парадоксальных следствия. Во-первых, давайте рассмотрим формализацию неформальной теории множеств в виде теории первого порядка. Для этого достаточно просто переписать наши неформальные постулаты на языке формул; это множество формул обозначим SET и добавим его в качестве аксиом к AX; получилась аксиоматическая система AX(SET). Тогда если получившаяся система противоречива, то в ней доказуемо всё что угодно и тогда возникает естественный вопрос: зачем мы её изучаем? А если эта система непротиворечива, то, согласно теореме Лёвенгейма – Сколема, она имеет счётную модель. Но ведь в теории множеств доказывается существование бесконечных несчётных множеств: как же эти бесконечные несчётные множества «ухитряются» помещаться внутри счётных моделей? Парадокс!

На самом деле этот парадокс кажущийся. Ведь речь идёт о теории и моделях первого порядка. В таком случае нет разделения на элементы, множества элементов, множества множеств элементов и т. д., а все элементы модели – это объекты абсолютно одного порядка; точно так же, значки « \in » и « \subseteq » не интерпретируются «быть элементом» и «быть подмножеством», а могут интерпретироваться просто бинарными отношениями между объектами. Единственное требование – должны удовлетворяться все аксиомы SET. Ну а раз в такой модели «множества» – это просто объекты, то и «равномощно» или «неравномощно» – это просто некоторые отношения между этими объектами, а вовсе не реальные отношения между множествами.

Во-вторых, рассмотрим теорию первого порядка поля вещественных чисел. В качестве термов будут выступать выражения, построенные из символов «0», «1» и переменных при помощи символов двуместных операций «+», «-», « \times » и «:», т. е. все так называемые рациональные выражения. В качестве элементарных формул выступают равенства и неравенства рациональных выражений. Формулы же получаются обычным образом из элементарных за счёт отрицания, конъюнкций, дизъюнкций, квантификации. В таком случае теория первого порядка поля целых чисел – это множество T всех предложений описанного языка, которые истинны в модели (R, I), где R – множество всех действительных чисел, а интерпретация означает «0», «1», «+», «-», « \times », «:», «=», « \neq », « \leq » и « $<$ » естественным образом. За-

метим, что теория T – полная: для всякого предложения или оно, или его отрицание истинно в модели \mathbf{R} .

Тогда рассмотрим следующее предложение ϕ :

$$\forall a \forall b \forall c \forall d \exists z (a \times z^3 + b \times z^2 + c \times z + d = 0),$$

где z^3 и z^2 – это просто сокращения для $z \times z \times z$ и для $z \times z$ соответственно. Это предложение утверждает, что уравнение третьей степени с параметрическими коэффициентами обязательно имеет корень. Поэтому оно истинно в \mathbf{R} , так как достаточно заметить, что функция $(\lambda x \in \mathbf{R}. (a \times x^3 + b \times x^2 + c \times x + d))$ непрерывна, а $a \times (-\infty)^3 + b \times (-\infty)^2 + c \times (-\infty) + d$ и $a \times (+\infty)^3 + b \times (+\infty)^2 + c \times (+\infty) + d$ имеют разные знаки. Следовательно, предложение ϕ входит в теорию первого порядка поля вещественных чисел.

Теперь рассмотрим следующее семейство Ψ замкнутых формул:

$$\{ \exists a \exists b \exists c \exists d (\bigwedge_{t \in T} a \times t^3 + b \times t^2 + c \times t + d \neq 0) : T - \text{конечное множество рациональных выражений, построенных из } 0, 1 \text{ и переменных } a, b, c \text{ и } d \}.$$

Если множество предложений $\{ \phi \} \cup \Psi \cup T$ совместно, все предложения из этого множества истинны в Эрбрановой модели для $\{ \phi \} \cup \Psi \cup T$. Предложение ϕ утверждает существование корня для параметрического уравнения третьей степени, а предложения Ψ утверждают, что никакой замкнутый терм не обращает это параметрическое уравнение в верное равенство. Но в Эрбрановом универсуме все элементы – это просто термы. Противоречие. Следовательно, множество предложений $\{ \phi \} \cup \Psi \cup T$ несовместно. Тогда по теореме компактности найдётся конечное множество термов T , построенных из 0, 1 и переменных a, b, c и d , такое, что

$$\forall a \forall b \forall c \forall d (\bigwedge_{t \in T} a \times t^3 + b \times t^2 + c \times t + d = 0)$$

является утверждением теории T . Другими словами: корень любого уравнения третьей степени над полем действительных чисел представим в виде одного из фиксированного конечного множества рациональных выражений, построенных с использованием коэффициентов этого уравнения. Таким образом, мы «решили» кубическое уравнение общего вида без операции извлечения корня.

Вообще говоря, аналогичные рассуждения применимы к параметрическому уравнению любой нечётной степени, а значит, мы научились находить решение таких уравнений явно, в виде рациональных выражений. Но, как известно, уравнения степени выше четвертой не имеют решения в радикалах, т. е. представимого в виде рационального выражения в котором разрешается ещё использовать операцию извлечения корней. Парадокс!

На самом деле никакого парадокса нет, а просто была допущена ошибка в рассуждениях, когда мы обосновывали несовместимость множества предложений $\{ \phi \} \cup \Psi \cup T$ ссылкой на то, что все элементы Эрбранова

универсума – это термы. При этом мы «забыли», что в Эрбрановом универсуме все элементы – это замкнутые термы с новыми константами.

Лекция 5. Основы автоматического доказательства теорем

Верифицирующий транслятор и автоматическое доказательство теорем

Теперь всё готово к более или менее точной постановке проблемы верифицирующего транслятора в данном курсе.

Вход верифицирующего транслятора – вычислительные программы на языке НеМо, аннотированные (снабжённые) пред- и постусловиями, заданными в виде логических формул³¹.

Возможными выходами верифицирующего транслятора являются:

- сообщения о синтаксических ошибках в программе и/или условиях;
- сообщения о семантическом несоответствии программы аннотациям;
- исполнение программы на вычислительной машине, если не обнаружено синтаксических ошибок и семантических несоответствий.

Разумеется, можно желать от верифицирующего транслятора значительно большей функциональности. Например, вместо модельного языка НеМо использовать какой-либо реальный язык программирования или допустить хорошую локализацию ошибок и несоответствий. Но цель курса – изучить основы трансляции и верификации программ, поэтому мы ограничиваемся только описанной функциональностью, тем более что уже на этом уровне возможно почувствовать всю сложность и комплексность задачи.

Из сказанного следует, что для верифицирующего транслятора умение обрабатывать логические формулы столь же важно, как и «традиционное» для трансляторов умение находить синтаксические ошибки. Обо всём спектре того, что должен уметь делать верифицирующий транслятор с логическими формулами, пойдёт речь в курсе далее, но, само собой разумеется, что он должен уметь верифицировать логические формулы, т. е. иметь средства для «проверки» тождественной истинности и реализуемости.

Направление науки программирования, которое занимается изучением и реализацией на компьютерах средств «проверки» тождественной истинности и/или реализуемости логических формул называется автоматическим доказательством теорем. Основы автоматического доказательства теорем

³¹ Резидентные программы на языке НеМо, аннотированные (снабжённые) условиями безопасности, прогресса и/или справедливости, заданными в виде логических формул, не будут обсуждаться в этом учебном пособии.

заложили ещё в 50-е гг. XX в. М. Дэвис, Г. Логман и Д. Лавлэнд³². В этой лекции мы познакомимся с некоторым минимумом понятий из области автоматического доказательства теорем, необходимых для данного курса³³. Для того, чтобы изложение было более конкретным, будем излагать основные понятия на примере пропозициональной логики.

В таком случае у нас в распоряжении имеется метод проверки тождественной истинности посредством составления таблицы истинности формулы. Метод состоит в том, что перебираются все означивания всех атомов формулы и для каждого означивания вычисляются значения всех подформул и формулы; если при всех означиваниях значение формулы «И», то формула является тождественно истинной, иначе – не является. Данный метод является разрешающей процедурой, т. е. по каждой формуле за конечное число шагов позволяет заключить, является ли данная формула тождественно истинной или нет. Например, для следующей формулы

$$\begin{array}{c}
 \Phi \\
 \underbrace{\hspace{10em}} \\
 (\neg p \wedge q) \vee (\neg \neg(p \vee q) \wedge \neg(\neg p \wedge q)) \\
 \begin{array}{ccc}
 \Phi_1 & \underbrace{\hspace{4em}} & \Phi_1 \\
 & \Phi_2 &
 \end{array} \\
 \underbrace{\hspace{10em}} \\
 \Phi_3
 \end{array}$$

таблицу истинности можно представить таким образом:

p	q	¬p	Φ ₁	¬Φ ₁	Φ ₂	¬Φ ₂	¬¬Φ ₂	Φ ₃	Φ
И	И	Л	Л	И	И	Л	И	И	И
И	Л	Л	Л	И	И	Л	И	И	И
Л	И	И	И	Л	И	Л	И	Л	И
Л	Л	И	Л	И	Л	И	Л	Л	Л

Из этой таблицы следует, что Φ не является тождественно истинной формулой.

Достоинства данной разрешающей процедуры, как и вообще любой разрешающей процедуры, очевидны: для любой формулы она в принципе

³² По-видимому первая в широкой печати статья об опыте разработки и реализации автоматического доказателя теорем – это работа названных авторов: Davis M., Logemann G., Loveland D. A machine program for theorem-proving // Communications of the ACM. 1962. Vol.5. P.394 – 397.

³³ При этом ряд важнейших понятий не будет затронут вовсе (например, так называемый метод резолюций). Поэтому тем, кто заинтересован в более глубоком изучении основ данной отрасли науки программирования, можно рекомендовать фундаментальную монографию: Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.

позволяет получить ответ за конечное число шагов. Но у разрешающих процедур вообще и у рассмотренного метода таблиц истинности в частности есть и существенные недостатки. Их два. Во-первых, практически любая разрешающая процедура имеет высокую алгоритмическую сложность; в частности, метод таблиц истинности обязательно перебирает все 2^n возможных означивания для формулы с n различными атомами, что для больших n может быть огромным числом. Во-вторых, если реализованная на компьютере разрешающая процедура находит контрпример для какой-либо формулы (в случае пропозициональной логики это такое означивание, при котором значение формулы Л), то корректность такого ответа проверить обычно просто (для пропозициональной логики достаточно подставить значения, которые получают атомы при означивании в формулу, и вычислить соответствующее значение формулы); но если реализованная процедура сообщает о тождественной истинности формулы, то единственный вариант проверки корректности – запуск на этой же формуле другой реализации данной или другой разрешающей процедуры (так называемое «поливариантное программирование»). Заметим, что с высокой алгоритмической сложностью разрешающих процедур можно пытаться как-то бороться при помощи оптимизаций алгоритма, представления данных, оптимизацией программного кода. Но проблема надёжности (достоверности) реализации, вообще говоря, решается или поливариантным программированием, или средствами, отличными от разрешающих процедур.

Обсудим обе эти возможности повышения надёжности для пропозициональной логики. В качестве альтернативы методу таблиц истинности рассмотрим метод бинарных разрешающих деревьев, а в качестве альтернативы разрешающим процедурам – методы поиска доказательства в аксиоматической системе.

Бинарные разрешающие деревья

Алгоритм бинарных разрешающих деревьев

\\ (Первый пример алгоритма в нашем курсе.)

[$n \geq 0$ – целое число, p_0, \dots, p_n – различные атомы,

ϕ - пропозициональная формула, построенная из атомов p_1, \dots, p_n]

// Предусловие.

– Выполнить следующие шаги:

- а) построить полное бинарное дерево T высоты $(n + 1)$;
- б) пометить корень T формулой ϕ ;
- с) пометить все левые исходящие дуги в T значением «И»;
- д) пометить все правые исходящие дуги в T значением «Л».

- для всех k от 1 до $(n+1)$, для каждой вершины T на высоте k выполнить следующие шаги:
 - a) пусть $v \in \{И, Л\}$ – пометка входной дуги в эту вершину, а ψ – формула-пометка родителя этой вершины;
 - b) пусть ξ – формула, которая получается из ψ в результате подстановки вместо атома $p_{(k-1)}$ значения v ;
 - c) упрощать формулу ξ до тех пор пока применимо хотя бы к одной её подформуле хотя бы одно из следующих правил переписывания:

$$\begin{aligned} И \wedge \theta &\Rightarrow \theta, \quad \theta \wedge И \Rightarrow \theta, \\ Л \wedge \theta &\Rightarrow Л, \quad \theta \wedge Л \Rightarrow Л, \\ Л \vee \theta &\Rightarrow \theta, \quad \theta \vee Л \Rightarrow \theta, \\ И \vee \theta &\Rightarrow И, \quad \theta \vee И \Rightarrow И, \\ \neg И &\Rightarrow Л, \quad \neg Л \Rightarrow И \end{aligned}$$
 (где θ – произвольная пропозициональная формула);
 - d) пометить вершину формулой ξ .
- Если есть лист T , который помечен «Л», то
 - a) пусть $v_0, \dots, v_n \in \{И, Л\}$ – пометки дуг пути из корня T к этому листу;
 - b) $\text{counter_example} := \{(p_0, v_0), \dots, (p_n, v_n)\}$;
 - c) $\text{answer} := \text{«НЕТ»}$.
- Если все листья T помечены «И», то $\text{answer} := \text{«ДА»}$.
 [
 $\text{answer} = \text{«ДА»}$ и ϕ является тождественно истинной формулой, или
 $\text{answer} = \text{«НЕТ»}$ и ϕ не является тождественно истинной формулой,
 а counter_example – означивание атомов, при котором ϕ принимает значение Л
] // Постусловие.

Утверждение 1. Алгоритм бинарных разрешающих деревьев является totally корректным по отношению к своим пред- и постусловиям.

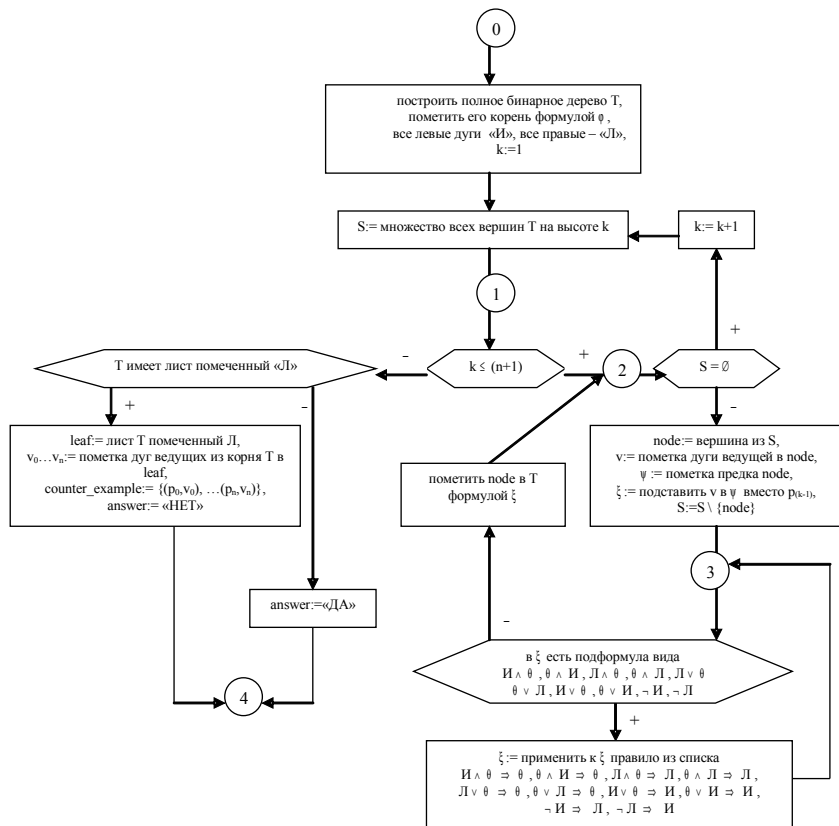
Доказательство. Идея доказательства проста: пути в полном бинарном дереве от корня к вершинам дерева определяют означивание части атомов, а пути к листьям – означивания всех атомов; метод бинарных разрешающих деревьев фактически состоит в последовательной подстановке в формулу ϕ вместо атомов, которые уже получили значения, самих этих значений и в упрощении получившихся формул. Мы, однако, проведём это дока-

зательство по с использованием методов Флойда³⁴. Для этого необходимо доказать частичную корректность нашего алгоритма и его завершаемость.

Завершаемость алгоритма следует из двух простых наблюдений:

7. цикл в пункте 2 перебирает все вершин полного бинарного дерева T ;
8. цикл в пункте 2.3 всё время сокращает размер формулы ξ .

Для доказательства частичной корректности необходимо сначала построить блок-схему алгоритма (см. ниже), определить контрольные точки и их инварианты.



³⁴ В данном курсе мы будем широко применять метод Флойда для доказательства алгоритмов.

Для удобства описания инвариантов определим три вспомогательных свойства:

- Complete Boolean Tree $cbt(T, m)$ означает, что T – полное бинарное дерево высоты m , в котором все левые дуги помечены «И», а все правые – «Л»;
 - Boolean Constants Free $bcf(\xi)$ (где ξ – формула) означает, что ξ не содержит подформул вида $I \wedge \theta, \theta \wedge I, L \wedge \theta, \theta \wedge L, L \vee \theta, \theta \vee L, I \vee \theta, \theta \vee I, \neg I$ или $\neg L$ (где θ – произвольная формула);
 - « $\xi \equiv (u_0/q_0 \dots u_m/q_m)$ » (где ξ и θ – формулы, u_0, \dots, u_k – значения из $\{I, L\}$, а q_0, \dots, q_k – атомы) означает, что ξ получена из θ в результате подстановки u_0 вместо $q_0, \dots, u_{(h-1)}$ вместо $q_{(h-1)}$ и выполнения упрощающих правил переписывания для подформул $I \wedge \theta \Rightarrow \theta, \theta \wedge I \Rightarrow \theta, L \wedge \theta \Rightarrow L, \theta \wedge L \Rightarrow L, L \vee \theta \Rightarrow \theta, \theta \vee L \Rightarrow \theta, I \vee \theta \Rightarrow I, \theta \vee I \Rightarrow I, \neg I \Rightarrow L$ и/или $\neg L \Rightarrow I$ (где θ – произвольная пропозициональная формула).
- Тогда инварианты точек 0 – 4 можно выразить следующим образом:

6. предусловие;
7. условие $1 \leq k \leq (n + 2)$ в конъюнкции со следующими условиями:
 - 7.1. $cbt(T, n + 1)$ и корень T помечен формулой φ ;
 - 7.2. S – множество всех вершин T на высоте k ;
 - 7.3. любая вершина T на высоте $h \in [0..(k - 1)]$ помечена некоторой формулой ψ такой, что $bcf(\psi)$ и $\psi \equiv \varphi(v_0/p_0 \dots v_{(h-1)}/p_{(h-1)})$, где $v_0 \dots v_{(h-1)}$ – путь от корня T к этой вершине;
8. условие $1 \leq k \leq (n+2)$ в конъюнкции со следующими условиями
 - 8.1. $cbt(T, n+1)$ и корень T помечен формулой φ ;
 - 8.2. S – некоторое множество вершин T на высоте k ;
 - 8.3. любая вершина T на высоте $h \in [0..(k - 1)]$ помечена некоторой формулой ψ такой, что $bcf(\psi)$ и $\psi \equiv \varphi(v_0/p_0 \dots v_{(h-1)}/p_{(h-1)})$, где $v_0 \dots v_{(h-1)}$ – путь от корня T к этой вершине;
 - 8.4. любая вершина T на высоте k , которая не входит в S , помечена некоторой формулой ψ такой, что $bcf(\psi)$ и $\psi \equiv \varphi(v_0/p_0 \dots v_{(k-1)}/p_{(k-1)})$, где $v_0 \dots v_{(k-1)}$ – путь от корня T к этой вершине;
9. условие $1 \leq k \leq (n + 2)$ в конъюнкции со следующими условиями
 - 9.1. $cbt(T, n + 1)$ и корень T помечен формулой φ ;
 - 9.2. $node$ – некоторая вершина T на высоте k , S – некоторое множество вершин T на высоте k , и $node \notin S$;
 - 9.3. любая вершина T на высоте $h \in [0..(k - 1)]$ помечена некоторой формулой ψ такой, что $bcf(\psi)$ и $\psi \equiv \varphi(v_0/p_0 \dots v_{(h-1)}/p_{(h-1)})$, где $v_0 \dots v_{(h-1)}$ – путь от корня T к этой вершине;

- 9.4. любая вершина T на высоте k , которая не входит в S и отлична от node , помечена некоторой формулой ψ , что $\text{bscf}(\psi)$ и $\psi \equiv \varphi(v_0/p_0 \dots v_{(k-1)}/p_{(k-1)})$, где $v_0 \dots v_{(k-1)}$ – путь от корня T к этой вершине;
- 9.5. $\xi \equiv \varphi(v_0/p_0 \dots v_{(k-1)}/p_{(k-1)})$, где $v_0 \dots v_{(k-1)}$ – путь от корня T к этой вершине node ;
10. постусловие.

Теперь нам необходимо выделить все ациклические пути между контрольными точками. Таких путей 8: $(0 \dots 1)$, $(1 + 2)$, $(2 + 1)$, $(2 - 3)$, $(3 + 3)$, $(3 - 2)$, $(1 - + 4)$, $(1 - - 4)$. Остаётся доказать, что для каждого из этих путей имеет место следующее свойство: если выполнено условие, сопоставленное началу пути, то выполнено условие, сопоставленное его концу.

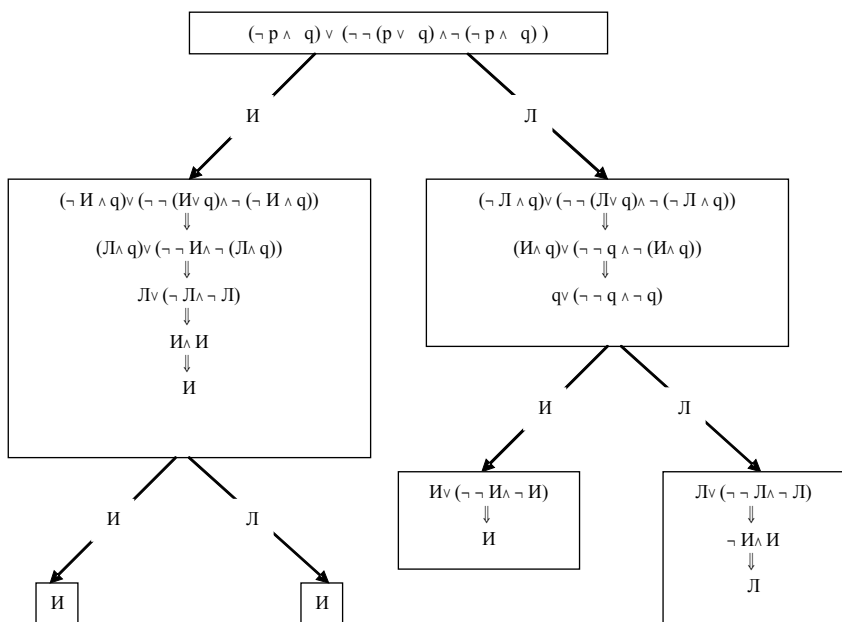
Для пути $(0 \dots 1)$ это свойство очевидно, так как в конце него $k = 1$. Для пути $(1 + 2)$ это свойство также имеет место, поскольку значение множества S не изменяется. Для пути $(2 + 1)$ свойство имеет место, так как $S = \emptyset$, и, следовательно, свойство $(2c)$ и $(2d)$ для k влекут свойство $(1c)$ для $(k+1)$. Путь $(2 - 3)$ обладает желаемым свойством очевидным образом. Свойство имеет место и для пути $(3 + 3)$, поскольку на нём просто применяется одно из разрешённых правил к формуле ξ . Для пути $(3 - 2)$ свойство следует из того, что разрешённые упрощающие правила более не применимы.

Доказательство путей $(1 - + 4)$ и $(1 - - 4)$ можно провести следующим образом. Предположим, что в начале этих путей имеет место инвариант (1) . Так как в начале них $k > (n + 1)$, то из $1 \leq k \leq (n + 2)$ следует, что $k = (n + 2)$. Следовательно, $(1c)$ влечёт, что все листья T помечены формулами, получившимися в результате упрощения φ в процессе подстановки вместо атомов значений из $\{I, L\}$, поэтому листья T помечены пропозициональными формулами, построенными из I и L без использования атомов. Но для каждой из этих формул в силу свойства $(1c)$ имеет место свойство bscf . Это означает, что всякая такая формула есть или I , или L (иначе нарушено свойство bscf). Кроме того, свойство $(1a)$ означает, что все $2^{(n+1)}$ набора значений атомов рассмотрены. Следовательно, в конце обоих путей имеет место постусловие 4. ■

Существенным недостатком метода бинарных разрешающих деревьев является явное построение полного бинарного дерева, поэтому позже мы рассмотрим, как обойти его без явного построения (т. е. оставить виртуальным) и возможности для оптимизации обходов.

Закончим обсуждение метода бинарных разрешающих деревьев разбором примера построения такого дерева для уже верифицированной формулы $(\neg r \wedge q) \vee (\neg \neg (p \vee q) \wedge \neg (\neg r \wedge q))$. На следующем рисунке вершины дерева помечены цепочками упрощающих правил, заданных в пункте 2.3 алгоритм-

ма; последняя формула каждой вершины – её пометка в соответствии с алгоритмом.



Методы поиска доказательства

В силу утверждения 1 из лекции 4 задача проверки тождественной истинности пропозициональной формулы равносильна проблеме доказуемости этой формулы во введенной в той же лекции надёжной и полной аксиоматизации АХ пропозициональной логики. Возникает стратегическая дилемма: в каком порядке пытаться конструировать деревья доказательств – сверху вниз или снизу вверх:

- сверху вниз начиная с аксиом генерировать все заключения, которые можно получить из ранее сгенерированных посылок, многократно применяя все правила вывода, пока не будет сгенерирована целевая формула;
- снизу вверх, начиная с целевой формулы, генерируются все посылки, из которых можно получить за однократное применение какого-либо

из правил вывода хотя бы одно из ранее сгенерированных заключений, пока не будут сгенерированы аксиомы.

Методы поиска доказательства, реализующие стратегию сверху вниз, называются прямыми, а методы, реализующие стратегию снизу вверх, – обратными.

Огромное достоинство систем, реализующих тот или иной метод поиска доказательства, по сравнению с разрешающими процедурами состоит в том, что, когда для формулы доказательства найдено, его проверка – это простая синтаксическая процедура обхода этого дерева и проверки, что все вершины – правила вывода, а все листья – аксиомы. Главный недостаток систем поиска доказательства состоит в том, что обычно нельзя предсказать возможный размер дерева доказательства и, следовательно, оценить, когда во время работы системы можно прекращать дальнейший поиск, так как доказательства не существует.

Познакомимся с общими проблемами, возникающими в прямых и обратных методах поиска доказательства. Для того, чтобы некоторые из обсуждаемых проблем были более конкретными и понятными, будем использовать ту же пропозициональную формулу ϕ , что и выше.

Прямые методы.

Аксиоматическая система может содержать бесконечное множество аксиом, заданных посредством схем аксиом. Перебирать всё это множество и строить все возможные доказательства до тех пор, пока не будет построено какое-либо доказательство целевой формулы, теоретически возможно. Но такой метод неэффективен и не позволяет за конечное число шагов сделать корректное заключение о недоказуемости этой формулы.

Поэтому первая тактическая проблема прямого вывода – это выбор конечного множества аксиом, из которых имеет смысл попытаться доказать целевую формулу. Для пропозициональной логики в рамках нашей аксиоматической системы AX к успеху приводит тактика использования аксиом, где заключение состоит только из тех атомов и/или их отрицаний, которые встречаются в доказываемой формуле. В частности, для формулы ϕ – это множество аксиом, у которых заключение $\Sigma \subseteq \{p, q, \neg p, \neg q\}$.

После того как при помощи некоторой тактики выбрано конечное множество аксиом, можно попытаться генерировать множество всех заключений, доказуемых из данного множества аксиом. Так как выбрано конечное множество аксиом, то на любом конечном шаге генерации множество доказанных формул и применимых правил вывода будет конечно.

Вторая тактическая проблема прямого вывода – это проблема ограничения размера заключений, доказуемых из выбранного множества аксиом.

Для пропозициональной логики в рамках нашей аксиоматической системы АХ к успеху приводит тактика генерации только тех заключений, в которых суммарное число связок \neg , \wedge и \vee не больше, чем в целевой формуле. В случае формулы ϕ ограничение числа связок 10.

Третья тактическая проблема прямого вывода – проблема ограничения множества правил вывода, которые нужно применять на каждом шаге. (Ведь если множество применимых правил вывода на каждом шаге состоит из n правил, то за k шагов может быть сгенерировано n^k заключений.) Для пропозициональной логики в рамках нашей аксиоматической системы АХ к успеху приводит тактика генерации тех заключений, которые состоят только из подформул целевой формулы. В нашем случае это множество правил вывода, у которых заключение $\Sigma \subseteq \{ p, q, \neg p, \phi_1, \neg \phi_1, \phi_2, \neg \phi_2, \neg \neg \phi_2, \phi_3, \phi \}$.

И, наконец, обсудим проблему восстановления доказательства для прямых методов. Так как прямые методы генерируют новые доказуемые заключения из аксиом и ранее доказанных посылок при помощи правил вывода, то самое естественное хранить вместе с каждым доказанным заключением это правило вывода и указатели на посылки, использованные в нем. Такая организация данных при применении прямых методов позволяет легко восстановить все дерево доказательства для любого доказуемого заключения и предъявить его для проверки.

Обратные методы.

Каждое конечное множество формул может быть заключением для нескольких правил вывода. В общем случае это множество правил вывода может быть вообще бесконечным. Перебирать всё это множество и строить все возможные деревья «доказательства» до тех пор, пока не будет построено какое-либо доказательство целевой формулы, в принципе, возможно. Но такой метод тоже не эффективен и не позволяет за конечное число шагов сделать корректное заключение о недоказуемости этой формулы.

Поэтому первая тактическая проблема обратного вывода – это ограничения на каждом шаге поиска доказательства целевой формулы конечного множества правил вывода, применимых на этом шаге. Для пропозициональной логики в рамках нашей аксиоматической системы АХ к успеху приводит тактика использования правил, содержащих по одному «упрощающему» правилу для каждой «составной» формулы (не атома или его отрицания), которая входит в рассматриваемое на данном шаге множество формул.

После того как при помощи некоторой тактики мы научились ограничивать на каждом шаге множество допустимых правил вывода, нам достаточно найти одно среди всех допустимых правил вывода, которое приводит

к доказательству целевой формулы. Но это правило вывода может быть многопосылочным, и нам необходимо доказать все его посылки. Это означает, что граф всех посылок, генерируемых при поиске доказательства целевой формулы, является $\forall\exists$ -графом с вершинами двух сортов:

- \exists -вершины, в которых происходит выбор одного из правил вывода;
- \forall -вершины, в которых инициализируется поиск доказательств всех посылок выбранного правила вывода.

Вторая тактическая проблема обратного вывода – это проблема восстановления доказательства, т. е. организации обхода $\forall\exists$ -графа всех посылок, сгенерированных при поиске доказательства целевой формулы: для \exists -вершины необходимо выбрать один из её наследников, а для \forall -вершины необходимо перебрать всех её наследников. Для пропозициональной логики в рамках нашей аксиоматической системы АХ это означает, что в случае выбора правила вывода для конъюнкции необходимо доказать обе посылки этого правила вывода.

Третья тактическая проблема обратного вывода – это проблема ограничения глубины обхода графа всех посылок. Для пропозициональной логики в рамках нашей аксиоматической системы АХ к успеху приводит тактика ограничения глубины числом связей \neg , \wedge и \vee в целевой формуле. В нашем случае ограничение глубины – 10.

Закончим неформальное обсуждение методов поиска доказательства ссылкой на две наиболее концептуально прозрачные системы автоматического поиска доказательства: это Otter для логики первого порядка (<http://www-unix.mcs.anl.gov/AR/otter/>) и HOL для логик высших порядков (<http://lsl.cs.byu.edu/lsl/hol-documentation.html>).

Лекция 6. Методы верификации пропозициональных формул

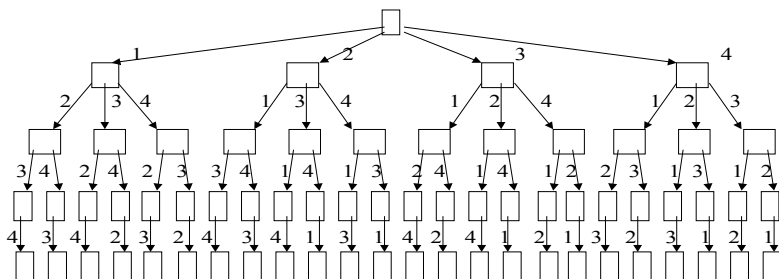
Обходы деревьев

Проблема обхода «виртуальных» деревьев (т. е. деревьев, которые не заданы явно, а строятся в процессе их обхода и не хранятся в памяти целиком) будет затрагиваться в данном курсе еще не раз. Например, при обсуждении метода бинарных разрешающих деревьев, методов поиска доказательства и проверки деревьев доказательства на корректность. Эта же проблема позже возникнет при трансляции НеМо-программ. Поэтому рассмотрим подробнее методы обходов таких деревьев (некоторые из которых могут оказаться потенциально бесконечными).

Обычно применяются два метода: метод отката и метод ветвей и границ. Оба метода имеют цель породить все листья (или вершины) дерева, удовлетворяющие некоторому целевому условию C . Общая идея обоих методов состоит в том, что кроме целевого условия C принимается во внимание так называемое граничное условие B , проверяемое в вершинах дерева во время обхода и являющееся достаточным что все листья, наследующие текущей вершине, не удовлетворяют целевому условию C . Разница методов состоит в порядке обхода вершин дерева:

- метод отката – это обход в глубину,
- метод ветвей и границ – это обход в ширину.

Продemonстрируем оба метода на известной задаче расстановки N ферзей на шахматной доске $N \times N$ так, чтобы ферзи не били друг друга. В обоих случаях последовательно генерируются частичные расстановки ферзей, которые не бьют друг друга. Начальная расстановка – это пустая доска; далее, если какая-то расстановка ферзей уже получена, то надо попытаться поставить нового ферзя, но сразу отбросить те варианты, в которых новый ферзь попадает под бой уже расставленных ферзей. В этой задаче дерево образуют, например, частичные расстановки ферзей, когда i -й ферзь ставится на i -ю строчку ($1 \leq i \leq N$), условие C на листья – N ферзей стоят на доске и не бьют друг друга, а граничное условие B – очередной i -й ферзь ($1 \leq i \leq N$) бьёт ранее поставленного ферзя с номером $j \in [1 .. (i-1)]$. Объяснять, как работают оба алгоритма, будем при $N = 4$, т. е. на доске 4×4 . В этом случае теоретически возможно следующее дерево всех расстановок ферзей по разным строчкам и столбцам, где дуги на высоте $i \in [1..4]$ помечены номерами колонок, в которые поставлен ферзь:



Алгоритм с откатом расстановки ферзей легче всего представить в форме с рекурсивной процедурой $NQ(i)$, осуществляющей поиск колонки k , куда на i -й строчке можно поставить i -го ферзя; эта процедура использует глобальную переменную $POS[1..N]$, в элементах которой $POS[1]$, ... $POS[i-1]$ хранятся номера колонок, где в 1-й, ... $(i-1)$ -й строчках поставлены ферзи с номерами 1, ... $(i-1)$:

VAR POS: ([1..N] ARRAY OF [1..N]) ;

PROCEDURE $NQ(i : INT)$;

{ FOR $k \in [1..N]$ DO

IF ферзь в позиции (i, k) не бьёт позиции $(1, POS[1])$, ... $((i-1), POS[i-1])$

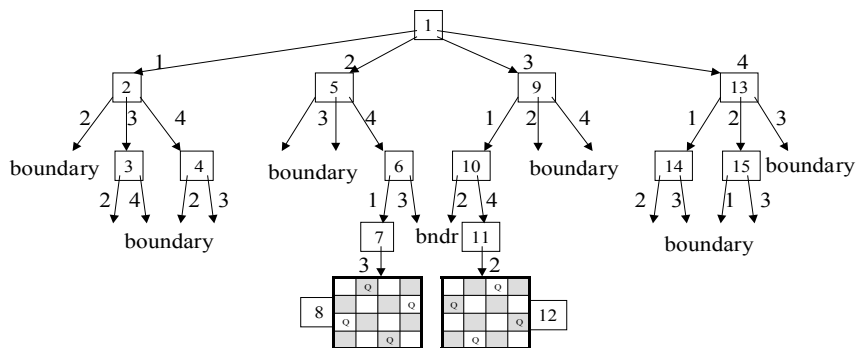
THEN { $POS[i] := k$;

IF POS содержит N ферзей THEN output(POS) ELSE $NQ(i+1)$ }

} ;

BEGIN $NQ(1)$ END .

Исполнение этого алгоритма «выльется» в следующий обход приведённого выше дерева всех расстановок ферзей:



Boundary (bndr): вновь поставленный ферзь бьёт некоторого ранее поставленного ферзя.

На этом рисунке числа рядом с дугами имеют тот же смысл, что и на предыдущем рисунке, а числа внутри вершин соответствуют порядку порождения вершин при обходе получающегося дерева.

Алгоритм ветвей и границ расстановки ферзей легче всего представить в итеративной форме с очередью:

```

VAR POS : ([1..N] ARRAY OF [1..N]) ;
VAR QUE : QUEUE OF ([1..N] ARRAY OF [1..N]) ;

BEGIN
POS := «пустая доска» ; QUE := <POS> ;
DO POS := head(QUE) ; QUE := tail(QUE) ;
  IF POS содержит N ферзей THEN output(POS)
  ELSE
  {i:= первая свободная строка позиции POS ;
  FOR k∈[1.. N] DO
    IF ферзь в позиции (i,k) не бьёт позиции (1,POS[1]), ... ((i-1),POS[i-1])
    THEN { POS[i] := k ; QUE := QUE^POS }
  }
UNTIL (QUE = <>)
END .

```

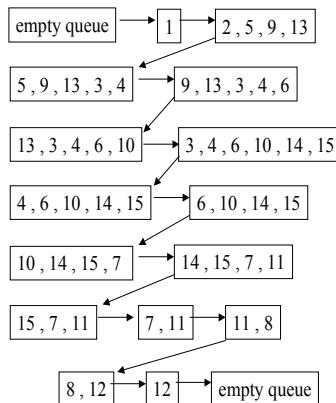
В этом алгоритме использованы следующие обозначения:

1. $\langle \rangle$ для пустой очереди;
2. $\langle x \rangle$ для одноэлементной очереди из x ;
3. $\langle x_1, x_2 \dots x_m \rangle$ для очереди из $m \geq 2$ элементов;
4. \wedge для операции добавления в хвост очереди;
5. head для операции взятия головы очереди;
6. tail для операции взятия хвоста очереди.

Все перечисленные операции не имеют побочного эффекта и обладают следующими свойствами:

- $\text{head}(\langle \rangle)$ и $\text{tail}(\langle \rangle)$ неопределены;
- $\text{head}(\langle x \rangle) = x$ и $\text{tail}(\langle x \rangle) = \langle \rangle$;
- $\text{head}(\langle x_1, x_2 \dots x_m \rangle) = x_1$ и $\text{tail}(\langle x_1, x_2 \dots x_m \rangle) = \langle x_2 \dots x_m \rangle$;
- $\langle \rangle \wedge y = \langle y \rangle$, $\langle x \rangle \wedge y = \langle x, y \rangle$ и $\langle x_1, x_2 \dots x_m \rangle \wedge y = \langle x_1, x_2 \dots x_m, y \rangle$.

Исполнение данного алгоритма «выльется» в следующую эволюцию очереди QUE (использованы номера вершин из предыдущего рисунка):



Таким образом, общая схема рекурсивного алгоритма обхода методом отката какого-либо «виртуального» дерева Т следующая:


```
VAR POS: (INT ARRAY OF (узел T)) ;
```

```
PROCEDURE BackTrack(i: INT)
{ FOR EACH child of POS[k-1] in T DO
  IF child не удовлетворяет граничному условию B
  THEN { IF child является листом T
        THEN { IF child удовлетворяет условию C THEN output (child) }
        ELSE BackTrack(i+1)
        }
  } ;
```

```
BEGIN BackTrack(1) END .
```

Общая же схема итеративного алгоритма с очередью обхода методом ветвей и границ какого-либо «виртуального» дерева T выглядит следующим образом:

```
CONST root := корень T ;
VAR node, child : узел T ; VAR QUE : очередь для узлов из T ;

BEGIN
QUE := <root> ;
DO node := head(QUE) ; QUE := tail(QUE) ;
  IF node является листом T
  THEN { IF node удовлетворяет условию C THEN output (node) }
  ELSE
  { FOR EACH child - наследника node, который не удовлетворяет условию B
    DO QUE := QUE ^ child }
  UNTIL (QUE = ⟨⟩)
END .
```

Метод Дэвиса – Путнам

Применим метод отката для оптимизации обхода «виртуального» бинарного разрешающего дерева. Для этого специализируем приведённый выше общий алгоритм. Узлы бинарного разрешающего дерева – это пропозициональные формулы, поэтому массив POS: (INT ARRAY OF (узлы T)) «превращается» в массив frm: ([0..(n+1)] ARRAY OF (пропозициональная_формула)), а граничное условие – это «формула, соответствующая вершине, есть И или Л». В результате получаем следующий

Алгоритм проверки тождественной истинны
 // методом обхода с откатом бинарного разрешающего дерева

```
[
  n ≥ 0 – целое число, p0, ... pn – различные атомы,
  φ – пропозициональная формула, построенная из атомов p0, ... pn
] // Предусловие.

VAR flag : BOOLEAN ;
VAR frm : ( [0 .. (n+1)] ARRAY OF <пропозициональные_формулы> ) ;

PROCEDURE BDT(i: INT)
VAR ξ : <пропозициональная_формула> ;
{ FOR EACH v ∈ {И, Л} DO
  { ξ := подставить v вместо p(i-1) в frm[i] ;
    упрощать формулу ξ пока применимо хотя бы к одной её подформуле
    хотя бы одно из следующих правилами переписывания:
    И ∧ θ ⇒ θ, θ ∧ И ⇒ θ, Л ∧ θ ⇒ Л, θ ∧ Л ⇒ Л, Л ∨ θ ⇒ θ,
    θ ∨ Л ⇒ θ, И ∨ θ ⇒ И, θ ∨ И ⇒ И, ¬И ⇒ Л, ¬Л ⇒ И
    (где θ - произвольная пропозициональная формула) ;
    IF ξ есть И или Л
    THEN { IF ξ есть Л THEN flag := FALSE }
    ELSE BDT(i+1)
    }
  } ;

BEGIN
flag := TRUE ; frm[0] := φ ; BDT(1) ;
IF flag THEN answer := «ДА» ELSE answer := «НЕТ»
END .

[
  answer = «ДА» и φ является тождественно истинной формулой, или
  answer = «НЕТ» и φ не является тождественно истинной формулой
] // Постусловие.
```

(Для вычисления контрпримера достаточно ввести дополнительную глобальную переменную типа [0..n] ARRAY OF {И, Л}, в которой хранятся значения локальной переменной v и которая изменяется которую только в

теле процедуры одновременно с изменением значения флага «flag := FALSE» в операторе «IF ξ есть Л THEN flag := FALSE».)

Описанный метод проверки тождественной истинности пропозициональных формул (так же как и метод бинарных разрешающих деревьев) легко трансформируется в метод проверки реализуемости пропозициональных формул: достаточно лишь в теле программы заменить инициализацию «flag := TRUE» на «flag := FALSE», а в теле процедуры – оператор «IF ξ есть Л THEN flag := FALSE» на «IF ξ есть И THEN flag := TRUE». Получается следующий

Алгоритм проверки реализуемости

// методом обхода с откатом бинарного разрешающего дерева

```
[
n ≥ 0 – целое число, p0, ... pn – различные атомы,
φ – пропозициональная формула, построенная из атомов p0, ... pn
] // Предусловие.
```

```
VAR flag : BOOLEAN ;
```

```
VAR frm: ( [0..(n+1)] ARRAY OF <пропозициональная_формула> ) ;
```

```
PROCEDURE SAT(i: INT)
```

```
VAR  $\xi$ : <пропозициональная_формула> ;
```

```
{ FOR EACH v ∈ {И, Л} DO
```

```
  {  $\xi$  := подставить v вместо p(i-1) в frm[i] ;
```

```
  упрощать формулу  $\xi$  до тех пор пока применимо хотя бы к одной её
  подформуле хотя бы одно из следующих правилами переписывания:
```

```
И ∧  $\theta$  ⇒  $\theta$ ,  $\theta$  ∧ И ⇒  $\theta$ , Л ∧  $\theta$  ⇒ Л,  $\theta$  ∧ Л ⇒ Л, Л ∨  $\theta$  ⇒  $\theta$ ,  $\theta$  ∨ Л ⇒  $\theta$ ,
```

```
И ∨  $\theta$  ⇒ И,  $\theta$  ∨ И ⇒ И, ¬И ⇒ Л, ¬Л ⇒ И (где  $\theta$  – произвольная
пропозициональная формула) ;
```

```
IF  $\xi$  есть И или Л
```

```
THEN { IF  $\xi$  есть И THEN flag := TRUE }
```

```
ELSE SAT(i+1)
```

```
}
```

```
} ;
```

```
BEGIN
```

```
flag := FALSE ; frm[0] := φ ; SAT(1) ;
```

```
IF flag THEN answer := «ДА» ELSE answer := «НЕТ»
```

```
END .
```

[
answer = «ДА» и ϕ является реализуемой формулой, или
answer = «НЕТ» и ϕ не является реализуемой формулой
] // Постусловие.

Заметим, что проверить достоверность ответа «ДА» программы, которая реализует этот алгоритм, значительно проще, чем проверить достоверность положительного ответа программы, реализующей любой метод проверки тождественной истинности: не нужно никакое поливариантное программирование, а достаточно просто найти пример означивания атомов, который даёт истинность формулы, подставить соответствующие значения в формулу и вычислить её значение. А для вычисления такого означивания в приведённый алгоритм метода проверки реализуемости достаточно ввести дополнительную глобальную переменную типа $([0..n] \text{ ARRAY OF } \{I, L\})$, в которой хранить значения локальной переменной v и которая изменяется только в теле процедуры одновременно с изменением значения флага «flag := TRUE» в операторе «IF ξ есть I THEN flag := TRUE».

Дальнейшая оптимизация методов с откатом проверки тождественной истинности и/или реализуемости пропозициональных формул возможна, если известна дополнительная информация о самой формуле. Особое место среди таких специализированных методов принадлежит методу, разработанному и реализованному М. Дэвисом и Х. Путнам³⁵ для проверки реализуемости пропозициональных формул в так называемой конъюнктивной нормальной форме. Определим эту форму и сам метод проверки³⁶.

Синтаксис конъюнктивных нормальных формул (кнф) – то есть формул в конъюнктивной нормальной форме – определяется исходя из \langle атомов \rangle (то есть замкнутых элементарных формул), TRUE и FALSE (вместо которых можно использовать И и Л) следующим образом.

³⁵ Речь идёт об уже упоминавшейся первой системе автоматического доказательства. Заметим, однако, что не смотря на прошедшие полвека, этот метод остаётся самым производительным и востребованным.

³⁶ Обсуждение того, что всякая пропозициональная формула эквивалентна некоторой формуле в конъюнктивной нормальной форме, а также быстрый метод преобразования любой пропозициональной формулы к равнореализуемой формуле в конъюнктивной нормальной форме, выходит за рамки данного курса. Заинтересованным читателям можно порекомендовать любой учебник по пропозициональной логике, например: Мендельсон Э. Введение в математическую логику. М.: Наука, 1976. Гл. 1. Исчисление высказываний; Ершов Ю.Л., Палютин Е.А., Математическая логика. М.: Наука, 1987. Гл. 1 Исчисление высказываний.

Определение 1.

$\langle \text{литерал} \rangle^{37} ::= \langle \text{атом} \rangle \mid \neg \langle \text{атом} \rangle$

$\langle \text{клауз} \rangle ::= \text{TRUE} \mid \text{FALSE} \mid \langle \text{литерал} \rangle \mid \langle \text{клауз} \rangle \vee \langle \text{клауз} \rangle$

$\langle \text{кнф} \rangle ::= (\langle \text{клауз} \rangle) \mid \langle \text{кнф} \rangle \wedge \langle \text{кнф} \rangle$

Семантика кнф определяется на моделях обычным образом с учётом следующих соображений. Как известно, для всякой ассоциативной алгебраической двухместной операции \oplus порядок выполнения не имеет значения: $(\dots((a_1 \oplus a_2) \oplus a_3) \oplus \dots a_n) = (\dots(a_1 \oplus (a_2 \oplus a_3)) \oplus \dots a_n) = \dots = (a_1 \oplus (a_2 \oplus (a_3 \dots \oplus a_n) \dots))$; поэтому $a_1 \oplus a_2 \oplus a_3 \dots \oplus a_n$ можно определить как общее значение всех этих алгебраических выражений. Так как семантика конъюнкции и дизъюнкции – ассоциативные операции пересечения и объединения множеств, то семантика многоместных конъюнкций (в кнф) и многоместных дизъюнкций (в клаузах) определяется обычным образом как пересечение и объединение.

А в терминах означиваний семантику кнф можно определить так.

Определение 2. Пусть выбрано и зафиксировано означивание атомов посредством И и Л:

- клауз означивается как И тогда и только тогда, когда он содержит TRUE или хотя бы один литерал, означенный как И, иначе он означивается как Л;
- кнф означивается как И тогда и только тогда, когда все её клаузы означены как И, иначе кнф означивается как Л.

Основная идея метода Дэвиса – Путнам состоит в использовании так называемых юнитов³⁸. Юнит – это клауз, состоящий из одного литерала. Если кнф ξ при некотором означивании принимает значение И, то любой юнит α в этой кнф при этом означивании должен принимать значение И. Следовательно, литерал β , соответствующий юниту α , тоже должен принимать значение И при данном означивании. Поэтому, если этот литерал – некоторый атом γ , то он должен означиваться И, а если литерал – отрицание $\neg\gamma$ некоторого атома, то этот атом γ должен означиваться как Л. Означивание атома γ , означивающее соответствующий литерал β и юнит α как И, называется правильным для юнита α : правильное значение атома γ для юнита (γ) есть И, а для юнита $(\neg\gamma)$ есть Л. В результате получаем следующий

Алгоритм М. Дэвиса и Х. Путнам

// проверки реализуемости пропозициональных формул

[

³⁷ От англ. literal и clause – «буквенный» и «параграф».

³⁸ От англ. unit – неделимая единица.

$n \geq 0$ – целое число, p_0, \dots, p_n – различные атомы,
 ϕ - конъюнктивная нормальная формула, построенная из атомов p_1, \dots, p_n
] // Предусловие.

VAR flag : BOOLEAN ;
 VAR frm: ([0 .. (n+1)] ARRAY OF <пропозициональные_формулы>) ;

PROCEDURE PD_SAT(i : INT)

VAR ξ : <пропозициональная_формула> ;

{ FOR EACH $v \in \{И, Л\}$ DO

{DO

$\xi :=$ подставить v вместо $p_{(i-1)}$ в $frm[i]$;

упрощать формулу ξ пока применимо хотя бы к одной её

подформуле хотя бы одно из следующих правилами переписывания:

$И \wedge \theta \Rightarrow \theta, \theta \wedge И \Rightarrow \theta, Л \wedge \theta \Rightarrow Л, \theta \wedge Л \Rightarrow Л, Л \vee \theta \Rightarrow \theta, \theta \vee Л \Rightarrow \theta,$

$И \vee \theta \Rightarrow И, \theta \vee И \Rightarrow И, \neg И \Rightarrow Л, \neg Л \Rightarrow И$ (где θ - произвольная
 пропозициональная формула) ;

IF ξ содержит юнит(ы)

THEN { $\psi :=$ юнит в ξ ; $v :=$ правильное означивание атома для ψ }

UNTIL ξ содержит юнит(ы) ;

IF ξ есть И или Л

THEN { IF ξ есть И THEN flag := TRUE }

ELSE PD_SAT(i+1)

}

} ;

BEGIN

flag := FALSE ; frm[0] := ϕ ; PD_SAT(1) ;

IF flag THEN answer := «ДА» ELSE answer := «НЕТ»

END .

[

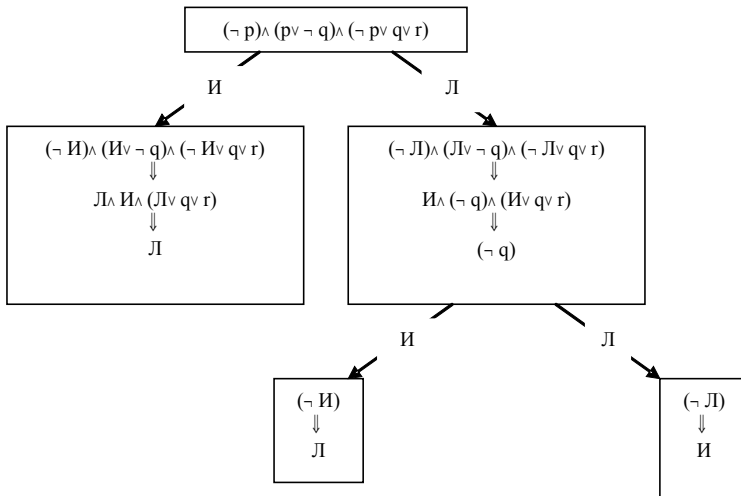
answer = «ДА» и ϕ является реализуемой формулой, или

answer = «НЕТ» и ϕ не является реализуемой формулой

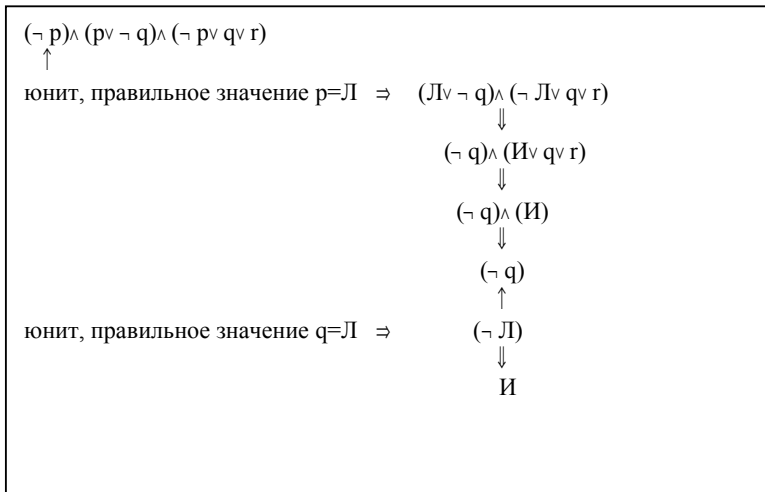
] // Постусловие.

Проиллюстрируем эффективность метода Дэвиса – Путнам на примере формулы $(\neg p) \wedge (p \vee \neg q) \wedge (\neg p \vee q \vee r)$. Алгоритм проверки реализуемости будет

обрабатывать эту формулу следующим образом, совершая 5 вызовов процедуры SAT:



А алгоритм Х. Путнам и М. Дэвиса обработает эту формулу во время первого и единственного вызова процедуры PD_SAT следующим образом:



Лабораторная работа 1.

Алгоритмы верификации Формул пропозициональной логики

Цель лабораторной работы: возобновление навыков работы с формулами пропозициональной логики и практическое знакомство с:

- машинно-ориентированными стандартами представления формул пропозициональной логики;
- алгоритмами проверки тождественной истинности и реализуемости формул пропозициональной логики.

В лабораторную работу входит:

1. Знакомство с форматом sat и cnf представления формул пропозициональной логики.
2. Уточнение, дизайн, реализация и тестирование алгоритма проверки реализуемости методом Дэвис – Путнам с генерацией примера для реализуемых формул.
3. Уточнение, дизайн, реализация и тестирование алгоритма проверки тождественной истинности методом отката с генерацией контрпримера для формул, которые не являются тавтологиями. (Дополнительно, по желанию.)

Этапы:

- Знакомство с форматом sat и cnf представления формул пропозициональной логики. Дизайн, реализация и тестирование конвертора (преобразователя) формата cnf в sat.
- Уточнение, дизайн, реализация и индивидуальное тестирование алгоритма
 1. проверки реализуемости конъюнктивных нормальных формул методом Дэвис – Путнам (с генерацией примера для реализуемых формул);
 2. проверки тождественной истинности пропозициональных формул методом отката (с генерацией контрпримера для формул, которые не являются тавтологиями).
- Взаимное тестирование (кросс-тестирование) реализаций алгоритмов проверки тождественной истинности методом отката и проверки реализуемости методом Дэвис – Путнам (с использованием конвертора cnf в sat).

(Пункты 2.2 и 3 являются дополнительными и выполняются по желанию.)

Описание форматов sat и cnf.

Описанные ниже входные машинно-ориентированные форматы представления формул пропозициональной логики разработаны и поддержива-

ются центром дискретной математики и программирования DIMACS³⁹. Основная цель предложенных форматов – обеспечить унифицированный ввод данных и возможность сравнивать по производительности верификаторы и решатели пропозициональных формул⁴⁰.

Формат `cnf` служит для представления конъюнктивных нормальных формул. Файл состоит исключительно из ASCII-символов и разбивается на две логические части: преамбулу и тело.

Преамбула `cnf`-формата содержит неформальную и формальную информацию о формуле и состоит из нескольких строчек. Сначала идут строчки неформальных комментариев, каждая из которых начинается с латинской буквы *c*, за которой следует пробел. Заканчивается преамбула одной обязательной строкой с формальной информацией о формуле, начинающейся с латинской буквы *p*, за которой следует пробел и три поля $\langle \text{формат} \rangle$, $\langle \text{число_атомов} \rangle$ и $\langle \text{число_кловов} \rangle$, разделённые пробелами:

$\langle \text{формальная_информация} \rangle ::= p \langle \text{формат} \rangle \langle \text{число_атомов} \rangle \langle \text{число_кловов} \rangle$.

Для `cnf`-формата значение поля $\langle \text{формат} \rangle$ всегда «`cnf`», а полей $\langle \text{число_атомов} \rangle$ и $\langle \text{число_кловов} \rangle$ – целые положительные числа в десятичной записи, представляющие число различных атомов и общее число кловов в формуле. Предполагается, что все атомы специфицируемой конъюнктивной нормальной формулы – это p_1, \dots, p_n , где n – число, специфицированное полем $\langle \text{число_атомов} \rangle$. Предполагается также, что всего кловов в специфицируемой конъюнктивной нормальной формуле m , где m – число, специфицированное полем $\langle \text{число_кловов} \rangle$.

Тело `cnf`-формата представляет клозы конъюнктивной нормальной формулы. Каждый литерал представляется целым числом в десятичной записи: атомы p_1, \dots, p_n представляются числами $[1..n]$, а отрицания атомов $\neg p_n, \dots, \neg p_1$ – числами $[-n .. -1]$. Каждый кюз представляется перечислением представлений его литералов (скобки «(» и «)»), а также все знаки дизъюнкции « \vee » опускаются. Знак конъюнкции « \wedge » между (между!) клозами представляется числом 0. Разделителями служат пробелы, табуляция и переводы строки.

Пример `cnf`-формата. Конъюнктивная нормальная формула $(p_1 \vee p_3 \vee \neg p_4) \wedge (p_4) \vee (p_2 \vee \neg p_3)$ может быть представлена ASCII-файлом со следующим содержанием:

³⁹ Discrete Mathematics and Theoretical Computer Science – см. URL <http://dimacs.rutgers.edu/index.html>.

⁴⁰ В частности, формат `cnf` используется для проведения соревнований решателей конъюнктивных нормальных формул – см. URL <http://www.satlive.org/SATCompetition/index.jsp>.

с Это пример cnf-формата.

p cnf 4 3

1 3–4 0

4 0 2

–3

Формат sat служит для представления произвольных пропозициональных формул, построенных из литералов, в которых конъюнкция « \wedge » и дизъюнкция « \vee » могут иметь какое угодно число аргументов⁴¹. Также как и формат cnf, файл состоит исключительно из ASCII-символов и разбивается на две логические части: преамбулу и тело.

Преамбула cnf-формата содержит неформальную и формальную информацию о формуле и состоит из нескольких строчек. Сначала идут строчки неформальных комментариев, каждая из которых начинается с латинской буквы *c*, за которой следует пробел. Заканчивается преамбула одной обязательной строкой формальной информации о формуле, начинающейся с латинской буквы *p*, за которой следует пробел и два поля <формат> и <число_атомов>, разделённые пробелами:

<формальная_информация> ::= p <формат> <число_атомов>.

Для sat-формата значение поля <формат> всегда «sat», а поля <число_атомов> – целое положительное число в десятичной записи, представляющее число различных атомов в формуле. Предполагается, что все атомы специфицируемой формулы – это p_1, \dots, p_n , где n – число, специфицированное полем <число_атомов>.

Тело sat-формата представляет формулу следующим образом: атомы p_1, \dots, p_n – числами $[1..n]$, а отрицания атомов $\neg p_n, \dots, \neg p_1$ – числами $[-n \dots -1]$; далее представление формул определяется по индукции:

1. $\text{sat}(\text{TRUE}) = *()$,
2. $\text{sat}(\text{FALSE}) = +()$,
3. $\text{sat}(\neg(\varphi)) = -(\text{sat}(\varphi))$,
4. $\text{sat}(\varphi_1 \wedge \dots \varphi_m) = *(\text{sat}(\varphi_1) \dots \text{sat}(\varphi_m))$,
5. $\text{sat}(\varphi_1 \vee \dots \varphi_m) = +(\text{sat}(\varphi_1) \dots \text{sat}(\varphi_m))$.

⁴¹ Конъюнкция нуля аргументов понимается как TRUE, дизъюнкция 0 аргументов понимается как FALSE, конъюнкция и дизъюнкция одного аргумента совпадает с этим аргументом, конъюнкция и дизъюнкция двух аргументов определяются стандартным образом, а многоместные конъюнкция и дизъюнкция определяются также как для конъюнктивных нормальных формул.

В качестве разделителей в формате sat там, где это возможно (например, чтобы облегчить чтение) или необходимо (например, в списках аргументов конъюнкций и дизъюнкций), могут использоваться пробелы, табуляция и переводы строки. Кроме того, sat-формат разрешает использовать дополнительные парные скобки «(...)» вокруг представления любой формулы.

Пример sat-формата. Уже знакомая нам конъюнктивная нормальная формула $(p_1 \vee p_3 \vee \neg p_4) \wedge (p_4) \vee (p_2 \vee \neg p_3)$ может быть представлена в sat-формате следующим ASCII-файлом:

с Это пример sat-формата.

```
p sat 4
*(+(1 3 -4)
+(4)
+(2 -3)))
```

Требования «заказчика» к описанию конвертора формата cnf в sat:

1. Оба формата cnf и sat должны быть описаны в нотации Бэкуса–Наура.
2. Итеративный алгоритм конвертора должен быть представлен в форме псевдокода или блок-схемы высокого уровня.
3. Алгоритм должен быть сопровождён небольшой пояснительной запиской об используемых структурах данных, операциях на них.

Общий объём описания cnf и sat, алгоритма и сопроводительной записки – 2–3 страницы. Блок-схема может быть нарисована от руки.

Требования «заказчика» к реализации конвертора формата cnf в sat:

1. Предусловие: входные данные – ASCII файл с расширением cnf, в котором записана входная конъюнктивная нормальная формула в формате cnf.
2. Постусловие: выходные данные – ASCII файл с расширением sat, в котором записана входная конъюнктивная нормальная формула в формате sat.

Требования «заказчика» к программе, реализующей метод Х. Путнам и М. Дэвиса:

1. Предусловие: входные данные – конъюнктивная нормальная формула с $n > 1$ атомами p_1, \dots, p_n , представленная в cnf-формате записанная во входном файле с расширением cnf.
2. Постусловие: выходные данные – ASCII файл с тем же именем, что и входной файл, но с расширением out содержит:
 1. единственный символ «0», если формула не реализуется;

2. символ «1», если формула реализуется, за которым (через пробел) следует n символов из «0» и/или «1» (тоже разделённые пробелом), представляющих значения Л и И атомов p_1, \dots, p_n , при которых формула принимает значение И.
3. Нефункциональное требование: программа должна соответствовать дизайну алгоритма Х. Путнам и М. Дэвиса, описанному в лекции.
Требования «заказчика» к программе, реализующей рекурсивный алгоритм обхода с откатом бинарного разрешающего дерева:
 1. Предусловие: входные данные – формула с $n > 1$ атомами p_1, \dots, p_n , представленная в sat-формате и записанная во входном файле с расширением с sat.
 2. Постусловие: Выходные данные – ASCII файл с тем же именем, что и входной файл, но с расширением out содержит:
 - 2.1. единственный символ «1», если формула является тавтологией;
 - 2.2. символ «0», если формула не является тавтологией, за которым (через пробел) следует n символов из «0» и/или «1» (тоже разделённые пробелом), представляющих значения Л и И атомов p_1, \dots, p_n , при которых формула принимает значение Л.
3. Нефункциональное требование: программа должна соответствовать дизайну рекурсивного алгоритма обхода с откатом бинарного разрешающего дерева, описанному в лекции.

Для желающих предлагается провести взаимное тестирование (крос-тестирование) в автоматическом режиме реализаций алгоритмов проверки тождественной истинности методом отката и проверки реализуемости методом Дэвиса–Путнам (с использованием собственного конвертора `cnf` в `sat`) на автоматически генерируемой представительной системе тестовых примеров.

Часть II. СИНТАКСИС

Лекция 7. Синтаксис языков программирования

Нотация Бэкуса – Наура и синтаксические диаграммы Вирта

Как уже было сказано в предыдущих лекциях, для определения синтаксиса языков программирования широко применяется нотация, предложенная Дж. Бэкусом и П. Науром.

Определение 1. Нотация Бэкуса – Наура используют зарезервированные метасимволы «::=» и «⟨⟩», читающиеся как «является» и «или», определяемые понятия, которые выделяются посредством парных угловых скобок «⟨⟩» и «⟩», и символы алфавита языка (которые называются терминальными символами). Каждое правило (определение) в этой нотации имеет вид

определяемое понятие ::= серия альтернатив разделённых «|», где каждая из альтернатив может содержать как определяемые понятия, так и терминальные символы, а в каждой альтернативе синтаксические элементы следуют в порядке их перечисления слева направо. Каждое определяемое понятие может иметь несколько определений, которые можно сгруппировать в единое определение, перечислив через «|» альтернативы. Кроме того, в правой части правил можно использовать вспомогательные (парные) метаскобки «[⟨⟩, ⟨⟩]» «{⟨⟩}» и «{⟩}»:

1. «[⟨⟩ и ⟨⟩]» означают, что часть определения, заключённая в эти скобки, может быть опущена,
2. «{⟨⟩} и {⟩}» означают что часть определения, заключённая в эти скобки, может быть повторена любое число раз (0, 1, 2 и т.д.).

Точный смысл нотации будет ясен тогда, когда будет определено понятие грамматики и порождаемого ею языка. А пока ограничимся одним примером. Понятие ⟨число⟩ в языке НеМо может быть определено⁴² посредством следующих трёх правил:

⟨число⟩ ::= [⟨знак⟩] ⟨цифра⟩ {⟨цифра⟩}

⟨знак⟩ ::= + | –

⟨цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⁴² Здесь и далее в качестве примеров будут приводиться некоторые варианты определений, но не единственно возможные. В частности, некоторые варианты и примеры могут даже противоречить друг другу.

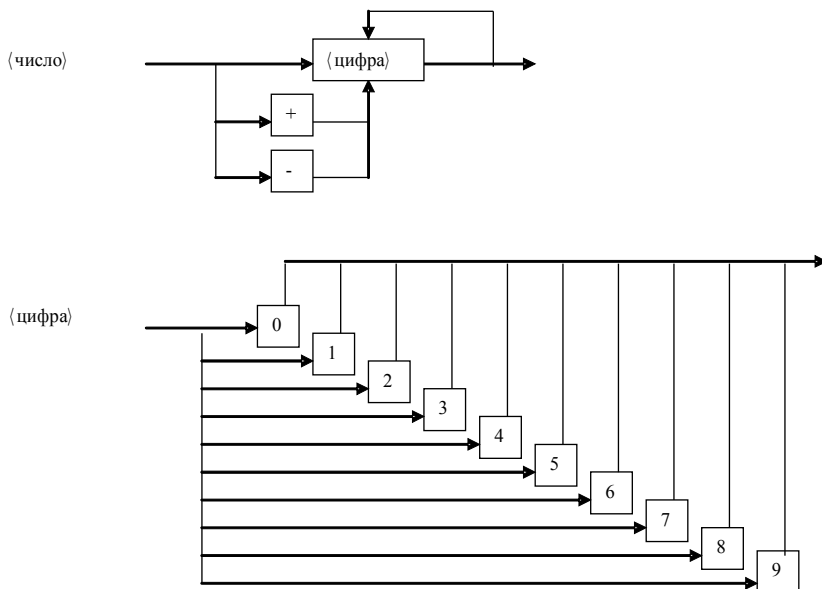
Согласно первому из этих правил, число – это любая конечная последовательность цифр, состоящая как минимум из одной цифры, которой может предшествовать знак. Согласно второму, знак – это плюс или минус. Согласно третьему, цифра – это десятичная цифра. В этом определении \langle число \rangle , \langle знак \rangle и \langle цифра \rangle – определяемые понятия, а знаки «+», «-» и цифры «0» ... «9» – терминальные символы. Недостаток данного определения состоит в том, что оно допускает возможность ведущих незначащих нулей: +00000 – вполне «законное» \langle число \rangle . Таким образом, \langle число \rangle , определённое по этим правилам, – это десятичное целое со знаком (возможно, с незначащими нулями).

Графический формализм для определения синтаксиса языков программирования – синтаксические диаграммы, введённые в активный «оборот» Н. Виртом.

Определение 2. В нотации синтаксических диаграмм каждое определение представляет собой ориентированный граф с одним входом и одним выходом, в котором вершины – это

- или терминальные символы,
 - или определяемые синтаксические понятия,
 - или разветвления/соединения для представления альтернатив,
- а дуги представляют отношение «следующий синтаксический элемент».

Пример синтаксических диаграмм: десятичные целые числа со знаком (с допустимыми незначащими нулями) могут быть определены следующей диаграммой.



Нотацию Бэкуса–Наура легко перевести в синтаксические диаграммы. Для этого, во-первых, надо «избавиться» от вспомогательных метаскобок «[» и «]», «{» и «}»:

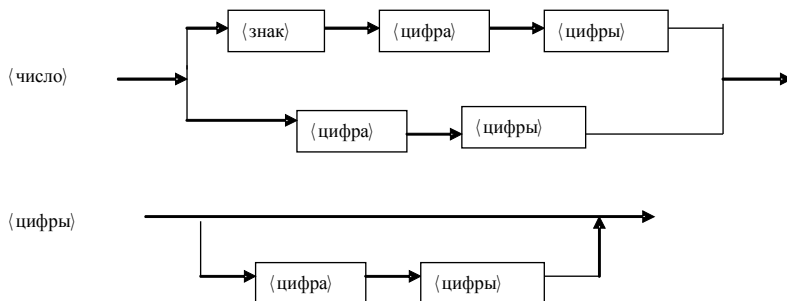
- всякое использование пары метаскобок «[» и «]» можно заменить парой альтернатив с использованием и без их содержимого;
- всякое использование пары метаскобок «{» и «}» можно заменить на новый $\langle \text{метаэлемент} \rangle$ и добавить новое определение $\langle \text{метаэлемент} \rangle ::= \langle \text{пусто} \rangle | \langle \text{содержимое метаскобок} \rangle \langle \text{метаэлемент} \rangle$ (где $\langle \text{пусто} \rangle$ имеет очевидное определение).

Во-вторых, надо каждое правило

определяемое понятие $::=$ серия альтернатив разделённых «|» заменить на синтаксическую диаграмму, которая начинается с ветвления по числу альтернатив и заканчивается соединением ветвей всех альтернатив, а каждая альтернатива представлена цепью, состоящей из элементов этой альтернативы, перечисленных в их естественном порядке слева направо.

Пример преобразования нотации Бэкуса – Наура в синтаксические диаграммы: определение $\langle \text{число} \rangle ::= [\langle \text{знак} \rangle] \langle \text{цифра} \rangle \{ \langle \text{цифра} \rangle \}$. Во-первых, избавимся от метаскобок:

- Во-вторых, заменим каждое из получившихся определений на диаграмму:



Синтаксические диаграммы также легко перевести в нотацию Бэкуса – Наура.

Грамматики, языки и классификация Н. Хомского

Глокая куздра штеко бодланула бокра и курдячит бокренка

Л. В. Щерба

Colorless green ideas sleep furiously

N. Chomsky

Определение 3. Грамматика – это четвёрка вида (N, T, R, S) , в которой N – конечный алфавит нетерминальных символов (или нетерминалов), T – конечный алфавит терминальных символов (или терминалов), R – конечное множество правил (или продукций) вида $w_L \rightarrow w_R$, где w_L и w_R – слова в объединённом алфавите $N \cup T$, а S – начальный (или стартовый) нетерминал.

Определение 4. Каждая грамматика $G = (N, T, R, S)$ определяет некоторый язык $L(G)$ следующим образом. Для любых слов α и β в объединённом алфавите $N \cup T$ будем:

- писать $\alpha \Rightarrow \beta$ и говорить, что « β получается из α в G за один шаг», если существуют такие слова w_1 и w_2 , w_L и w_R , что α можно предста-

в виде $w_1w_Lw_2$, β можно представить в виде $w_1w_Rw_2$, и $w_L \rightarrow w_R$ — одна из продукций из R ;

- писать $\alpha \Rightarrow_n \beta$ и говорить, что « β получается из α в G за n шагов» (где $n \geq 0$), если $n=0$ и α совпадает с β , или $n > 0$ и существует такое слово γ в объединённом алфавите $N \cup T$, что $\alpha \Rightarrow \gamma$ и $\gamma \Rightarrow_{(n-1)} \beta$;
- писать $\alpha \Rightarrow^* \beta$ и говорить, что « β получается из α в G », если существует такое $n \geq 0$, что $\alpha \Rightarrow_n \beta$.

Язык $L(G)$ состоит из всех слов α в алфавите терминальных символов T таких, что $S_G \Rightarrow^* \alpha$. Принято говорить, что грамматика G порождает язык $L(G)$.

Н. Хомский предложил следующую классификацию грамматик, языки которых не включают пустого слова (т. е. не содержащего ни одного символа).

Определение 5.

Регулярные грамматики используют продукции вида $n \rightarrow t$, $n \rightarrow tm$ и $n \rightarrow m$, где n и m — любые нетерминалы, а t — любой терминал.

Контекстно-свободные грамматики используют продукции вида $n \rightarrow w$, где n — любой нетерминал, а w — любое непустое слово в объединённом алфавите нетерминалов и терминалов.

Контекстно-зависимые (или неукорачивающие) грамматики используют продукции вида $w_L \rightarrow w_R$, где w_L и w_R — любые слова в объединённом алфавите нетерминалов и терминалов такие, что $|w_L| \leq |w_R|$.

Грамматики с фразовой структурой используют любые продукции с непустой правой частью.

Из определения следует, что классификация Хомского не задаёт разбиение всех грамматик на пересекающиеся классы, а представляет собой некоторую «матрёшку» из вложенных классов:



Определение 6. Классификация Н. Хомского распространяется на грамматики, языки которых включают пустое слово. Для грамматик с фразовой структурой разрешается использовать любые продукции. Для контекстно-зависимых грамматик разрешается использовать одно специальное правило $S \rightarrow \langle \text{пусто} \rangle$, где S - стартовый нетерминал. Для регулярных и контекстно-свободных грамматик являются допустимыми правила вида $a \rightarrow \langle \text{пусто} \rangle$ с любым метасимволом a .

Определение 7. Грамматики G' и G'' называются эквивалентными (по языку), если их языки совпадают: $L(G') = L(G'')$.

Сделаем следующее замечание об языковой эквивалентности регулярных и контекстно-свободных грамматик. Можно довольно-таки просто доказать, что всякая регулярная или контекстно-свободная грамматика G эквивалентна по языку некоторой грамматике того же класса, но которая, во-первых, не имеет так называемых «цепных правил», т. е. продукций вида $a \rightarrow m$, где a и m – произвольные нетерминалы; во-вторых, стартовый символ S не встречается в правых частях продукций; в третьих, все правила, кроме возможно одного, имеют непустую правую часть, а единственное допустимое правило с пустой правой частью – это $S \rightarrow \langle \text{пусто} \rangle$ (где S – стартовый нетерминал), допускается тогда и только тогда, когда $L(G)$ содержит пустое слово. Тем, кто заинтересован в доказательстве данного утверждения, рекомендуется обратиться к известным учебникам по синтаксическому анализу⁴³, например: Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции (М.: Мир, 1978. Т. 1 Синтаксический анализ, §2.4.2 Преобразования КС грамматик).

⁴³ Наш курс больше ориентирован на изучение семантических вопросов трансляции.

Определение 8. Язык L называется

- регулярным,
- контекстно-свободным,
- контекстно-зависимым,
- с фразовой структурой (или рекурсивно-перечислимым) если существует такая регулярная,
- контекстно-свободная,
- контекстно-зависимая,
- с фразовой структурой грамматика G , которая порождает этот язык L (т. е. $L = L(G)$).

Пример регулярного языка: десятичные целые числа со знаком (с допустимыми незначащими нулями). Пусть $G_{\text{int}} = (N_{\text{int}}, T_{\text{int}}, R_{\text{int}}, S_{\text{int}})$, где

- $N_{\text{int}} = \{S_{\text{int}}, D_{\text{int}}\}$ – алфавит метасимволов,
- $T_{\text{int}} = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ – алфавит терминальных символов,
- R_{int} есть объединение трёх следующих наборов продукций:
 - $S_{\text{int}} \rightarrow +D_{\text{int}}, S_{\text{int}} \rightarrow -D_{\text{int}}, S_{\text{int}} \rightarrow D_{\text{int}};$
 - $D_{\text{int}} \rightarrow 0D_{\text{int}}, D_{\text{int}} \rightarrow 1D_{\text{int}}, D_{\text{int}} \rightarrow 2D_{\text{int}}, D_{\text{int}} \rightarrow 3D_{\text{int}}, D_{\text{int}} \rightarrow 4D_{\text{int}}, D_{\text{int}} \rightarrow 5D_{\text{int}}, D_{\text{int}} \rightarrow 6D_{\text{int}}, D_{\text{int}} \rightarrow 7D_{\text{int}}, D_{\text{int}} \rightarrow 8D_{\text{int}}, D_{\text{int}} \rightarrow 9D_{\text{int}};$
 - $D_{\text{int}} \rightarrow 0, D_{\text{int}} \rightarrow 1, D_{\text{int}} \rightarrow 2, D_{\text{int}} \rightarrow 3, D_{\text{int}} \rightarrow 4, D_{\text{int}} \rightarrow 5, D_{\text{int}} \rightarrow 6, D_{\text{int}} \rightarrow 7, D_{\text{int}} \rightarrow 8, D_{\text{int}} \rightarrow 9.$

Ещё один пример регулярного языка: идентификаторы (т. е. последовательности букв и цифр, которые начинаются с буквы). Пусть $G_{\text{idn}} = (N_{\text{idn}}, T_{\text{idn}}, R_{\text{idn}}, S_{\text{idn}})$, где

- $N_{\text{idn}} = \{S_{\text{idn}}, D_{\text{idn}}\}$ – алфавит метасимволов,
- $T_{\text{idn}} = [a..z] \cup [0..9]$ – алфавит терминальных символов,
- R_{idn} есть объединение трех следующих наборов продукций:
 - $S_{\text{idn}} \rightarrow aD_{\text{idn}}, \dots S_{\text{idn}} \rightarrow zD_{\text{idn}};$
 - $D_{\text{idn}} \rightarrow aD_{\text{idn}}, \dots D_{\text{idn}} \rightarrow zD_{\text{idn}}, D_{\text{idn}} \rightarrow 0D_{\text{idn}}, \dots D_{\text{idn}} \rightarrow 9D_{\text{idn}};$
 - $D_{\text{idn}} \rightarrow a, \dots D_{\text{idn}} \rightarrow z, D_{\text{idn}} \rightarrow 0, \dots D_{\text{idn}} \rightarrow 9.$

Пример контекстно-свободного языка: арифметические выражения над целочисленными переменными и константами. Пусть $G_{\text{exp}} = (N_{\text{exp}}, T_{\text{exp}}, R_{\text{exp}}, S_{\text{exp}})$, где

- $N_{\text{exp}} = \{S_{\text{exp}}\} \cup N_{\text{int}} \cup N_{\text{idn}}$ – алфавит метасимволов,
- $T_{\text{exp}} = \{ (,) \} \cup T_{\text{int}} \cup T_{\text{idn}}$ – алфавит терминальных символов,
- R_{exp} есть объединение $R_{\text{int}}, R_{\text{idn}}$ и следующих трёх наборов продукций:
 - $S_{\text{exp}} \rightarrow S_{\text{int}}, S_{\text{exp}} \rightarrow S_{\text{idn}};$
 - $S_{\text{exp}} \rightarrow (S_{\text{exp}});$

– $S_{\text{exp}} \rightarrow S_{\text{exp}} + S_{\text{exp}}, S_{\text{exp}} \rightarrow S_{\text{exp}} - S_{\text{exp}}, S_{\text{exp}} \rightarrow S_{\text{exp}} * S_{\text{exp}}, S_{\text{exp}} \rightarrow S_{\text{exp}} / S_{\text{exp}} .$

Недостаток языка $L(G_{\text{exp}})$ – использование знака числа после открывающих скобок или после знака арифметической операции из-за правила $S_{\text{exp}} \rightarrow S_{\text{int}}$. Так, например, выражение $(+123 - +456 - -789 * -0)$ является вполне легитимным словом этого языка.

Контекстно-свободные грамматики и нотация Бэкуса – Наура

Пусть заданы совокупность определений⁴⁴ в нотации Бэкуса – Наура и выделено главное определяемое синтаксическое понятие. По этой совокупности определений и определяемому понятию можно определить контекстно-свободную грамматику (N, T, R, S) следующим образом. Алфавит нетерминалов N состоит из всех определяемых понятий (которые в соответствии с нотацией заключены в парные угловые скобки « $\langle \rangle$ » и « $\rangle \rangle$ »). Алфавит терминалов T совпадает с алфавитом терминальных символов, используемых в определениях. А совокупность продукций R получается в результате «разбиения» каждого определения

определяемое понятие ::= альтернатива₁ | ... | альтернатива_k

на серию из отдельных k определений

определяемое понятие ::= альтернатива₁,

...

определяемое понятие ::= альтернатива_k

и последующей замены метасимвола «является» «::=» на продукционную стрелку « \rightarrow »:

определяемое понятие \rightarrow альтернатива₁,

...

определяемое понятие \rightarrow альтернатива_k.

И, наконец, в качестве начального символа S примем главное определяемое понятие.

Контекстно-свободная грамматика, построенная по произвольной совокупности определений с выделенным главным определяемым понятием, и язык, порождаемый этой грамматикой, будем считать смыслом (семантикой) данной системы определений. Отметим, что при использовании нотации Бэкуса – Наура для определения какого-либо языка программирования главное понятие – это $\langle \text{программа} \rangle$ (которое обычно явно не выделяется, а подразумевается по умолчанию).

Заметим также, что любую контекстно-свободную грамматику легко представить в нотации Бэкуса – Наура: для этого достаточно вместо про-

⁴⁴ Как уже отмечалось ранее, не теряя общности, можно считать, что в определениях не используются вспомогательные метаскобки « $\langle \rangle$ », « $\rangle \rangle$ » « $\{ \}$ » и « $\} \}$ ».

дукционной стрелки « \rightarrow » использовать метасимвол « \vdash », заключить всякий нетерминал в угловые метаскобки « $\langle \rangle$ » и « \rangle », а начальный символ обвести выделенным главным определяемым понятием.

Регулярные языки

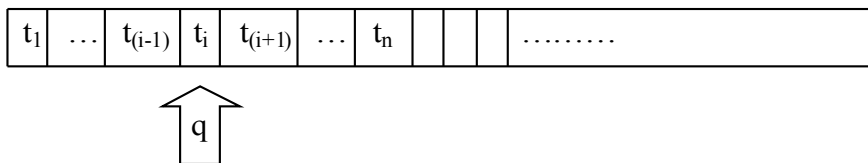
Определение 9. Конечный автомат – это пятёрка вида (Q, T, P, S, F) , где Q – конечное множество управляющих (контрольных) состояний, T – конечный алфавит входных (терминальных) символов, $P \subseteq Q \times T \times Q$ – программа (таблица) переходов, $S, F \in Q$ – стартовое (начальное) и финальное (заключительное) состояния.

Операционная семантика конечных автоматов определяется в терминах конфигураций, переходов между конфигурациями и языков, которые принимаются (распознаются) автоматами. Все эти понятия определяются ниже. Для удобства определения выберем и зафиксируем произвольный конечный автомат $A = (Q, T, P, S, F)$ и произвольное слово $\alpha = t_1 \dots t_n$ в алфавите T .

Определение 10. Конфигурация A состоит из:

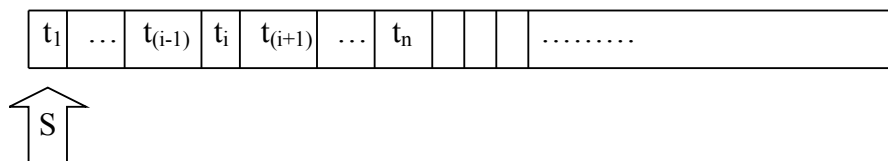
- 1) входной ленты, ограниченной слева и бесконечной вправо, разделённой на ячейки, на ней записано слово в алфавите T , по одному символу в ячейке, начиная с первой ячейки слева;
- 2) читающей головки, позиционированной в ячейке входной ленты, которая находится в управляющем состоянии из Q .

Графически конфигурация, в которой на входной ленте записано слово $t_1 \dots t_n$ в алфавите T , а читающая головка «обозревает» i -ую ячейку ленты ($1 \leq i \leq n + 1$) находится в состоянии $q \in Q$, которое принято изображать следующим образом:

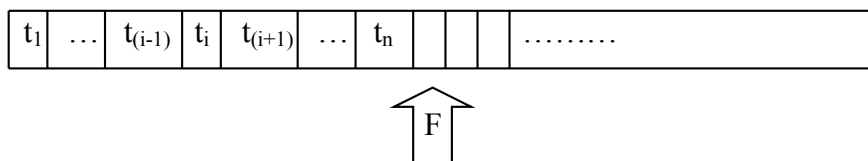


а текстуально эту конфигурацию принято представлять в виде тройки (q, i, α) .

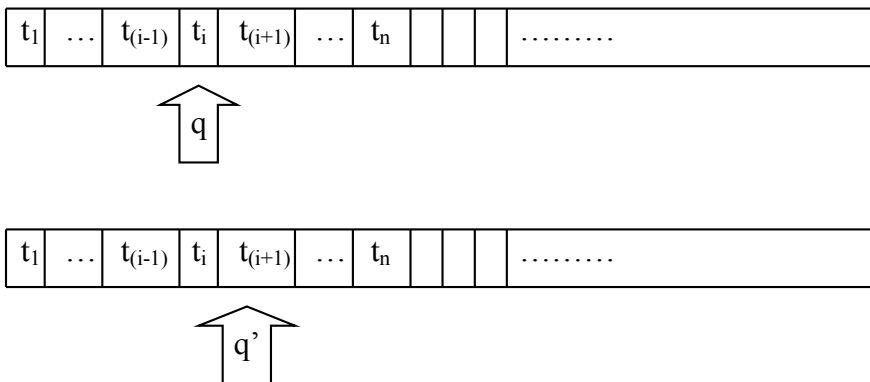
Определение 11. Начальная конфигурация – это конфигурация $(S, 1, \alpha)$, в которой головка обозревает левую ячейку ленты и находится в стартовом состоянии S :



Определение 12. Заключительная конфигурация – это конфигурация $(F, (n + 1), \alpha)$, в которой читающая головка обозревает первую пустую ячейку ленты и находится в финальном стоянии F :



Определение 13. Переход между конфигурациями – это произвольная пара конфигураций, такая, что $(q, t_i, q') \in P$ (см. рисунок ниже). В текстуальной форме данный переход принято записывать так: $(q, i, \alpha) \Rightarrow (q', (i+1), \alpha)$. В графической форме переход можно представить следующим образом:



Определение 14. Для произвольных конфигураций (q, i, α) и (q'', j, α) и произвольного $k \geq 0$ принято писать $(q, i, \alpha) \xrightarrow{A}_k (q'', j, \alpha)$ и говорить, что конфигурация (q'', j, α) получается из (q, i, α) за k шагов тогда и только тогда, когда $k = (i - j) = 0$ и $q = q''$ или когда $k = (j - i) > 0$ и существует такая конфигурация $(q', (i + 1), \alpha)$, для которой $(q, i, \alpha) \xrightarrow{A} (q', (i + 1), \alpha)$ и $(q', (i + 1), \alpha) \xrightarrow{A}_{(k-1)} (q'', j, \alpha)$. Другими словами, $(q, i, \alpha) \xrightarrow{A}_k (q'', j, \alpha)$ тогда и только тогда, когда существует последовательность из k конфигураций, в которой каждая следующая получается из предыдущей в результате перехода.

Определение 15. Для произвольных конфигураций (q, i, α) и (q'', j, α) принято писать $(q, i, \alpha) \xrightarrow{A}^* (q'', j, \alpha)$ и говорить, что конфигурация (q'', j, α) получается (достижима) из (q, i, α) тогда и только тогда, когда $(q, i, \alpha) \xrightarrow{A}_k (q'', j, \alpha)$ для некоторого $k \geq 0$.

Определение 16. Говорят, что автомат $A = (Q, T, P, S, F)$ распознаёт (принимает) слово $\alpha = t_1 \dots t_n$ в алфавите T тогда и только тогда, когда заключительная конфигурация со словом α достижима из начальной конфигурации с этим же словом на ленте, т. е. когда $(S, 1, \alpha) \xrightarrow{A}^* (F, (n+1), \alpha)$.

Для иллюстрации определений используем следующий конечный автомат ADD, в котором $Q_{\text{ADD}} = \{S, F, M\}$, $T_{\text{ADD}} = \{0, 1\} \times \{0, 1\} \times \{0, 1\}$, а программа P_{ADD} состоит из следующих переходов:

$(S, (0, 0, 0), S)$, $(S, (0, 1, 1), S)$, $(S, (1, 0, 1), S)$, $(S, (1, 1, 0), M)$,
 $(S, (0, 0, 0), F)$, $(S, (0, 1, 1), F)$, $(S, (1, 0, 1), F)$, $(S, (1, 1, 0), M)$,
 $(M, (0, 0, 1), S)$, $(M, (0, 1, 0), M)$, $(M, (1, 0, 0), M)$, $(M, (1, 1, 1), M)$, $(M, (0, 0, 1), F)$.

Допустимо использовать «табличный» формат для программы переходов, в котором на пересечении строки, соответствующей какому-либо со-

стоянию $q \in Q$, и столбца, соответствующего какому-либо символу алфавита $t \in T_{ADD}$, перечислены все такие q' , для которых $(q, t, q') \in P$:

	(0,0,0)	(0,0,1)	(0,1,0)	(0,1,1)	(1,0,0)	(1,0,1)	(1,1,0)	(1,1,1)
S	S, F			S, F		S, F	M	
M		S, F	M		M			M

Для большей наглядности будем изображать символы алфавита T_{ADD} не в строчку, а в столбик. Тогда любое слово в этом алфавите, записанное на входную ленту автомата ADD, можно мыслить как три двоичных числа с одинаковым количеством разрядов, записанных одно под другим. Например,

0	1	1	0	0	1	0	1	0	
1	1	1	1	0	0	0	0	1	
1	0	1	0	1	1	0	1	1	

представляет слово

$(0,1,1)(1,1,0)(1,1,1)(0,1,0)(0,0,1)(1,0,1)(0,0,0)(1,0,1)(0,1,1)$, а

0	1	0	1	0	1	0	1	0	
1	0	1	0	1	0	1	0	1	
0	0	0	0	0	0	0	0	0	

представляет

слово

$(0,1,0)(1,0,0)(0,1,0)(1,0,0)(0,1,0)(1,0,0)(0,1,0)(0,1,0)$.

Если предположить, что все эти три двоичных числа записаны «задом наперёд» (т. е. младшие разряды – слева, старшие – справа), тогда первый из приведенных примеров – правильный пример на сложение столбиком, соответствующий равенству 166 (первая строчка) + 271 (вторая строчка) = 437 (третья строчка). А второй пример – неправильный пример на сложение столбиком: 170 (первая строчка) + 341 (вторая строчка) $\neq 0$ (третья строчка).

Первый пример
 $(0,1,1)(1,1,0)(1,1,1)(0,1,0)(0,0,1)(1,0,1)(0,0,0)(1,0,1)(0,1,1)$
 принимается автоматом ADD, что подтверждается следующей
 «эволюцией» состояний вдоль слова:

0	1	1	0	0	1	0	1	0	
1	1	1	1	0	0	0	0	1	
1	0	1	0	1	1	0	1	1	



А вот второе слово
 $(0,1,0)(1,0,0)(0,1,0)(1,0,0)(0,1,0)(1,0,0)(0,1,0)(1,0,0)(0,1,0)$
 не принимается автоматом ADD, так как уже для первого символа этого
 слова $(0,1,0)$ не существует такого состояния $q \in Q_{ADD} = \{S, M, F\}$, что
 $(S, (0,1,0), q) \in P_{ADD}$.

То, что автомат ADD принимает первое слово и не принимает второе, неслучайно. Оказывается, для любого слова $\alpha = t_1 \dots t_n$ в алфавите T_{ADD} , для любого $i \in [1..n]$ имеет место следующее:

- 1) $(S, i, \alpha)_{ADD} \Rightarrow_{(n+1-i)} (F, (n+1), \alpha)$ тогда и только тогда, когда суффикс $t_i \dots t_n$ слова α правильный пример на сложение двоичных чисел, записанных задом наперёд;
- 2) $(M, i, \alpha)_{ADD} \Rightarrow_{(n+1-i)} (F, (n+1), \alpha)$ тогда и только тогда, когда суффикс $t_i \dots t_n$ слова α правильный пример на сложение числа 1 (единицы переноса) и двоичных чисел, записанных задом наперёд.

Доказательство легко провести индукцией по $i \in [1..n]$. Отсюда следует, в частности, что $(S, 0, \alpha)_{ADD} \Rightarrow^* (F, (n+1), \alpha)$ тогда и только тогда, когда α правильный пример на сложение двоичных чисел, записанных задом наперёд.

Определение 17. Пусть A – конечный автомат. Язык, который распознаёт (допускает или принимает) A , – это множество $L(A)$ всех слов, принимаемых этим автоматом.

Теорема 1 (о распознавании регулярных языков)

- Всякий регулярный язык допускается некоторым конечным автоматом.
- Всякий язык, который допускается конечным автоматом, является регулярным.

Доказательство.

1. Пусть L – произвольный регулярный язык. Пусть $G = (N, T, R, S)$ – регулярная грамматика, порождающая L , т. е. $L = L(G)$. Если язык L не содержит пустого слова, то можно считать, что все продукции этой грамматики имеют вид $n \rightarrow t$ и $n \rightarrow tm$, где n и m – любые нетерминалы, а t – любой терминал; если же язык L содержит пустое слово, то грамматика G содержит ещё одно правило $S \rightarrow \langle \text{пусто} \rangle$. Оба случая рассматриваются аналогично, поэтому рассмотрим только первый, когда пустое слово не содержится в языке L . В этом случае определим автомат A следующим образом. В качестве множества состояний Q примем множество нетерминалов N , расширенное заключительным состоянием F : $Q = N \cup \{F\}$. Начальное состояние автомата совпадает с начальным символом грамматики. Множество терминалов автомата, разумеется, совпадает с T . Множество продукций P состоит из двух частей:

- $\{ (n, t, m) : n, m \in N, t \in T \text{ и } (n \rightarrow tm) \in R \};$
- $\{ (n, t, F) : n \in N, t \in T \text{ и } (n \rightarrow t) \in R \}.$

Тогда для любого $k \geq 1$, любого слова $t_1 \dots t_k$ в алфавите T и любых $n_1, \dots, n_k \in N$ имеем: $n_1 G \Rightarrow t_1 n_2 G \Rightarrow \dots G \Rightarrow t_1 \dots t_{(k-1)} n_k G \Rightarrow t_1 \dots t_{(k-1)} t_k$ тогда и только тогда, когда $(n_1, 1, t_1 \dots t_k) A \Rightarrow (n_2, 2, t_1 \dots t_k) A \Rightarrow \dots (n_k, k, t_1 \dots t_k) A \Rightarrow (F, (k+1), t_1 \dots t_k)$. Действительно, для любого $i \in [1..(k-1)]$ имеем:

$$\begin{aligned}
 (n_i, i, t_1 \dots t_k) A &\Rightarrow (n_{(i+1)}, (i+1), t_1 \dots t_k) \\
 &\Downarrow \Uparrow \\
 (n_i, t_i, n_{(i+1)}) &\in P \\
 &\Downarrow \Uparrow \\
 (n_i \rightarrow t_i n_{(i+1)}) &\in R \\
 &\Downarrow \Uparrow \\
 t_1 \dots t_{(i-1)} n_i G &\Rightarrow t_1 \dots t_{(i-1)} t_i n_{(i+1)}.
 \end{aligned}$$

Аналогично

$$\begin{aligned}
 (n_k, k, t_1 \dots t_k) A &\Rightarrow (F, (k+1), t_1 \dots t_k) \\
 &\Downarrow \Uparrow \\
 (n_k, t_k, F) &\in P \\
 &\Downarrow \Uparrow \\
 (n_k \rightarrow t_k) &\in R \\
 &\Downarrow \Uparrow \\
 t_1 \dots t_{(k-1)} n_k G &\Rightarrow t_1 \dots t_{(k-1)} t_k.
 \end{aligned}$$

2. Пусть L – произвольный язык, допускаемый конечным автоматом. Пусть $A = (Q, T, P, S, F)$ – конечный автомат, который принимает этот язык. Определим регулярную грамматику G следующим образом. В ка-

честве алфавита нетерминалов N примем множество состояний Q . Разумеется, алфавит терминальных символов будет T , а начальным нетерминальным символом будет S . Множество productions R грамматики G состоит из двух подмножеств:

- $\{ n \rightarrow tm : n, m \in Q, t \in T \text{ и } (n, t, m) \in P \}$
- $\{ n \rightarrow t : n \in Q, t \in T \text{ и } (n, t, F) \in R \}$.

Тогда для любого $k \geq 1$, любого слова $t_1 \dots t_k$ в алфавите T и любых $q_1, \dots, q_k \in Q$ имеем: $q_1 \xrightarrow{G} t_1 q_2 \xrightarrow{G} \dots \xrightarrow{G} t_1 \dots t_{(k-1)} q_k \xrightarrow{G} t_1 \dots t_{(k-1)} t_k$ тогда и только тогда, когда $(q_1, 1, t_1 \dots t_k) \xrightarrow{A} (q_2, 2, t_1 \dots t_k) \xrightarrow{A} \dots (q_k, k, t_1 \dots t_k) \xrightarrow{A} (F, (k+1), t_1 \dots t_k)$. (Доказательство аналогично предыдущему случаю.)

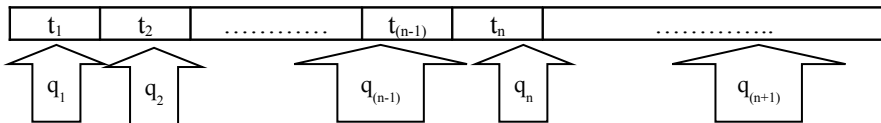
■

Следствием из этой теоремы является следующая «лемма о разрастании».

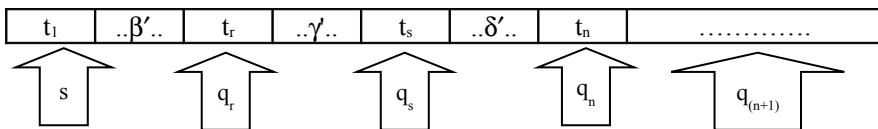
Теорема 2 (лемма о разрастании для регулярных языков). Для всякого регулярного языка L существует такое число $k > 0$, что любое слово $\alpha \in L$, длина которого $|\alpha|$ больше k , можно представить в виде $\beta\gamma\delta$ (где γ непустое слово), так, что для любого $i > 0$ слово $\beta\gamma^i\delta \in L$ (где γ^i есть $\gamma \dots \gamma$).

i-раз

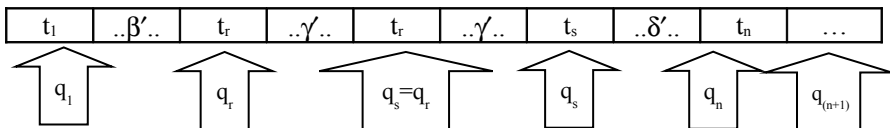
Доказательство. Пусть L – произвольный регулярный язык. Тогда существует некоторый конечный автомат $A = (Q, T, P, S, F)$, который принимает этот язык (т. е. $L = L(A)$). Пусть k число состояний автомата A (т. е. $K = |Q|$). Для любого слова $\alpha = t_1 \dots t_n$, длина которого $|\alpha| = n > k$ и которое принимает автомат A , существует последовательность состояний $q_1, \dots, q_{(n+1)}$ такая, что $q_1 = S$, $q_{(n+1)} = F$, а то, что слово α принимается автоматом A , подтверждается следующей «эволюцией» состояний вдоль ленты автомата:



Так как слово α имеет длину $|\alpha| > k$, то среди состояний $q_1, \dots, q_{(n+1)}$ найдётся хотя бы одно повторяющееся. Пусть q_s (где $s \in [1..k]$) – первое слева такое состояние, которое уже ранее встречалось, т. е. для которого $q_s = q_r$ (где $r \in [1..(s-1)]$). Имеем:



где β' , γ' и δ' – соответствующие части слова α . Так как $q_r = q_s$, то



представляет последовательность конфигураций, которая подтверждает, что слово $t_l\beta't_r\gamma't_r\gamma't_s\delta't_n$ принимается автоматом A и, следовательно, принадлежит языку L . Поэтому если принять $t_l\beta'$ в качестве β , $t_r\gamma'$ – в качестве γ , а $t_s\delta't_n$ – в качестве δ , то получается, что слово $\alpha = \beta\gamma\delta$, а слово $t_l\beta't_r\gamma't_r\gamma't_s\delta't_n = \beta\gamma\gamma\delta$. Применив к слову $\beta\gamma\gamma\delta$ изложенную выше процедуру выделения подслова γ повторно, получаем, что $\beta\gamma\gamma\gamma\delta \in L$, а многократно – что $\beta\gamma^i\delta \in L$ для любого $i > 0$. По построению γ непустое слово. ■

Как явствует из доказательства леммы о разрастании, её формулировка может быть усилена следующим образом. Для всякого регулярного языка L существует такое число $k > 0$, что любое слово $\alpha \in L$, длина которого $|\alpha|$ больше k , можно представить в виде $\beta\gamma\delta$, так, что $|\beta| < k$, $0 < |\alpha| < k$ и для любого $i > 0$ слово $\beta\gamma^i\delta \in L$. В такой «усиленной» форме лемма о разрастании может использоваться для того, чтобы доказывать нерегулярность некоторых языков.

Например, покажем, что ранее определённый язык арифметических выражений над целочисленными переменными и константами не является регулярным. для этого предположим противное, т. е. предположим, что язык $L(G_{\text{exp}})$ является регулярным. Тогда (согласно усиленной лемме о разрастании) существует такое число $k > 0$, что любое слово $\alpha \in L$, длина которого $|\alpha|$ больше k , можно представить в виде $\beta\gamma\delta$, так, что $|\beta| < k$, $0 < |\gamma| < k$ и для любого $i > 0$ слово $\beta\gamma^i\delta \in L$. Выберем $n > 2k$. Слово $(^n 0)^n$, состоящее из n открывающих скобок, нуля и n закрывающих скобок, является правильно построенным арифметическим выражением и принадлежит языку $L(G_{\text{exp}})$. Это слово, в частности, можно представить в виде $\beta\gamma\delta$, так, что $|\beta| < k$, $0 < |\alpha| < k$ и для любого $i > 0$ слово $\beta\gamma^i\delta \in L$. В силу того что $n > 2k$, заключаем, что γ непусто и состоит только из открывающих скобок и, следовательно, $(^n + |\gamma| 0)^n \in L(G_{\text{exp}})$. Но это слово не является правильно построенным арифметическим выражением из-за дисбаланса открывающих и закрывающих ско-

бок и, следовательно, $(\overset{n}{+} \mid \overset{0}{-})^n \notin L(G_{\text{exp}})$. Противоречие. Следовательно, предположение, что $L(G_{\text{exp}})$ – регулярный язык, неверно.

Лекция 8. Контекстно-свободные языки

Дерево синтаксического разбора

Определение 1. Пусть $G = (N, T, R, S)$ – контекстно-свободная грамматика. Синтаксическое дерево (в G) – это произвольное конечное ориентированное упорядоченное дерево, все вершины которого помечены символами алфавита $(N \cup T)$, такое, что:

- 1) все листья помечены символами терминального алфавита T или пустыми словами;
- 2) все внутренние вершины помечены символами нетерминального алфавита N , причём корень помечен стартовым символом S ;
- 3) для каждой вершины если $n \in N$ – её пометка, а $w_1, \dots, w_k \in N \cup T$ – перечисленные слева направо пометки непосредственных наследников этой вершины, то продукция $(n \rightarrow w_1 \dots w_k) \in R$.

Определение 2. Пусть G – контекстно-свободная грамматика, T_G – какое-либо синтаксическое дерево в G . Если «прочитать» слева направо пометки всех листьев T_G , то получится некоторое слово α в терминальном алфавите; в таком случае принято говорить, что данное синтаксическое дерево T_G является деревом грамматического разбора этого слова α (в грамматике G).

Теорема 1 (о грамматическом разборе) Для любой контекстно-свободной грамматики G и слова α в алфавите терминальных символов имеет место следующая эквивалентность: α имеет дерево грамматического разбора в G тогда и только тогда, когда это слово α порождается грамматикой G .

Доказательство. Пусть $G = (N, T, R, S)$ – контекстно-свободная грамматика. Сначала покажем, что всякое слово, которое имеет дерево грамматического разбора в G , порождается G . Затем покажем, что всякое слово, которое порождается в G , обязательно имеет дерево грамматического разбора в G . В обоих случаях нам понадобятся семейство вспомогательных грамматик $G_n = (N, T, R, n)$, индексированных нетерминальными символами $n \in N$.

- Пусть слово α имеет дерево грамматического разбора tr в G . Для любой нелистовой вершины i этого дерева пусть $tr(i)$ – поддереву с этой вершиной, $n(i) \in N$ – пометка этой вершины, а $\alpha(i)$ – слово в терминальном алфавите, которое получается, если «прочитать» слева направо пометки

всех листьев дерева $tr(i)$. Очевидно, что $tr(i)$ является деревом грамматического разбора $\alpha(i)$ в грамматике $Gn(i)$ для любой нелистой вершины i . Индукцией по $k \geq 1$ легко показать, что для любой нелистой вершины i дерева tr , если высота дерева $tr(i)$ меньше k , то $\alpha(i)$ порождается в грамматике $Gn(i)$.

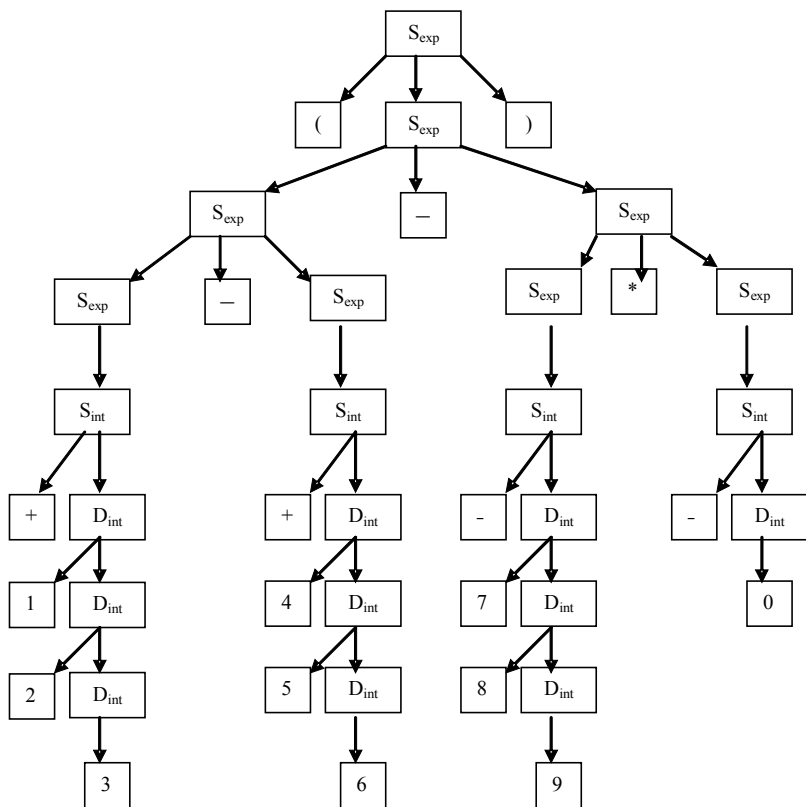
1. База индукции тривиальна, так как в дереве tr нет нелистовых вершин i , для которых высота поддерева $tr(i)$ меньше 1.
2. Предположение индукции: пусть для некоторого $k \geq 1$ утверждение верно.
3. Если дерево $tr(i)$ имеет высоту $(k+1)$, то пусть j_1, \dots, j_m — непосредственные наследники вершины i , а w_1, \dots, w_m — их пометки (всё перечислено в порядке слева направо). Все деревья $tr(j_1), \dots, tr(j_m)$ имеют высоту не больше k . Поэтому (в соответствии с предположением индукции) слова $\alpha(j_1), \dots, \alpha(j_m)$ порождаются в грамматиках $Gn(j_1), \dots, Gn(j_m)$. По определению синтаксического дерева, $(n(i) \rightarrow w_1 \dots w_m) \in R$. Следовательно, $n(i) \xrightarrow{G} w_1 \dots w_m \xRightarrow{*} \alpha(j_1) \dots \alpha(j_m) = \alpha(i)$, что и требовалось доказать.

Остаётся заметить, что если r — корень дерева, то $tr = tr(r)$, $S = n(r)$ и $\alpha = \alpha(r)$.

- Пусть слово α порождается в грамматике G , т. е. $S \xRightarrow{*} \alpha$. Индукцией по $k \geq 1$ легко показать, что для любого нетерминала $n \in N$ и любого слова β в терминальном алфавите T , если $n \xrightarrow{G} \beta$, существует дерево грамматического разбора β в грамматике Gn . Так как $S \xRightarrow{*} \alpha$ влечёт $S \xRightarrow{k} \alpha$ для подходящего $k \geq 1$, то в силу данного утверждения α имеет дерево грамматического разбора в G_S , т. е. в G .

■

Пример синтаксического дерева: арифметические выражения над целочисленными переменными и константами, порождаемые грамматикой $G_{\text{exp}} = (N_{\text{exp}}, T_{\text{exp}}, R_{\text{exp}}, S_{\text{exp}})$ (см. лекцию 7). Как уже отмечалось, «некрасивое» выражение $(+123 - +456 - -789 * -0)$ порождается этой грамматикой. Оно имеет следующее дерево грамматического разбора:



Определение 3. Контекстно-свободная грамматика в нормальной форме Хомского – это такая контекстно-свободная грамматика, в которой все productions (кроме, быть может, одной) имеют вид $n \rightarrow t$ или $m \rightarrow m'm''$, где t – терминал, n, m, m' и m'' – нетерминалы, стартовый символ не встречается в правых частях productions, а единственное допустимое правило с пустой правой частью – это $S \rightarrow \langle \text{пусто} \rangle$, где S – стартовый нетерминал.

Теорема 2 (о нормальной форме Н. Хомского) Всякая контекстно-свободная грамматика эквивалентна по языку контекстно-свободной грамматике в нормальной форме Хомского.

Доказательство (набросок) Пусть G – контекстно-свободная грамматика. В предыдущей лекции 7 было сделано замечание, что всякая контекстно-свободная грамматика эквивалентна по порождаемому языку некоторой контекстно-свободной грамматике, в которой нет цепных правил, стартовый символ не встречается в правых частях productions, а все productions, кроме, быть может, одного правила имеют непустую правую часть, а единственное (возможное) правило с пустой правой частью – это $S \rightarrow \langle \text{пусто} \rangle$ (где S – стартовый нетерминал). Поэтому мы сразу можем считать, что G обладает всеми этими свойствами. Следовательно, единственное что нам осталось сделать с G – это эквивалентным образом заменить «ненормализованные» productions вида $m \rightarrow w$, где m – нетерминал, а w – слово как минимум из двух символов, на productions, в которых правая часть – или одиночный терминал, или пара нетерминалов.

«Ненормализованные» productions можно устранить следующим образом. Пусть $m \rightarrow w$ одна из них и $k \geq 2$ – это длина слова w . Значит w – это некоторое слово $w_1w_2\dots w_k$, где все w_i ($i \in [1\dots k]$) – терминалы или нетерминалы G . Для каждого $i \in [1\dots k]$ определим символ W_i так:

$$W_i = \begin{cases} w_i, & \text{если } w_i \text{ – нетерминал } G, \\ \text{новый нетерминал } V_i, & \text{если } w_i \text{ – терминал } G. \end{cases}$$

Для каждого $i \in [2\dots(k-1)]$ определим новый нетерминал E_i . Теперь рассмотрим следующее множество productions:

$$\begin{aligned} m &\rightarrow W_1E_2, \\ E_2 &\rightarrow W_2E_3, \\ &\dots\dots\dots \end{aligned}$$

$$E_{k-2} \rightarrow W_{k-2} E_{k-1},$$

$$E_{k-1} \rightarrow W_{k-1} W_k.$$

Кроме того, для каждого $i \in [1..k]$, для которого w_i – терминал, добавим ещё одну продукцию $V_i \rightarrow w_i$. Теперь заменим продукцию $m \rightarrow w$ на все толь что введённые новые продукты и обозначим новую грамматику G' .

Тогда, с одной стороны, очевидным образом $m \xRightarrow{G'} w$, и, следовательно, $L(G) \subseteq L(G')$, а с другой – можно показать, что для любого слова α в алфавите терминалов и нетерминалов G' , если $m \xRightarrow{G'} \alpha$, $m \xRightarrow{G} \beta$, где β получается из α в результате замены всех E_i ($2 \leq i \leq (k-1)$) на $w_i \dots w_k$, а всех V_i ($1 \leq i \leq k$ и w_i – терминал) – на w_i ; поэтому $L(G') \subseteq L(G)$. Следовательно, $L(G) = L(G')$, но G' содержит меньше «ненормализованных» продукций. ■

Тому, кому интересны детали данного доказательства, рекомендуем обратиться к уже рекомендованному источнику: Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции (М.: Мир, 1978. Т. 1 Синтаксический анализ, §2.4.2 Преобразования КС грамматик).

Определение 4. Для любых двух слов α и β (некоторого алфавита) будем писать « $\beta \subseteq \alpha$ » и говорить, что « β является подсловом/подстрокой α », если $\alpha \equiv \delta \beta \gamma$ для некоторых слов γ и δ (т. е. α можно «разбить» на три последовательных части, одна из которых β). Отношение « \subseteq » на словах (некоторого алфавита) будем называть отношением включения.

Легко видеть, что отношение включения на словах любого алфавита является рефлексивным, транзитивным и антисимметричным. Поэтому оно является отношением частичного порядка⁴⁵.

Следовательно, любое конечное множество слов W можно «отсортировать» (т. е. перечислить без повторов) в виде последовательности w_0, \dots, w_n (где $n = |W|$ – количество элементов W) такой, что для любых $i, j \in [0..n]$, если $w_i \subseteq w_j$, то $i \leq j$. Для этого достаточно, например, просто «пере-

⁴⁵ Напомним основные определения для бинарных отношений. Пусть A – произвольное множество. Бинарное отношение $R \subseteq A \times A$ на множестве A называется:

- рефлексивным, если $(a, a) \in R$ для любого элемента $a \in A$;
- транзитивным, если для любых элементов $a, b, c \in A$ из $(a, b) \in R$ и $(b, c) \in R$ следует $(a, c) \in R$;
- симметричным, если для любых элементов $a, b \in A$ из $(a, b) \in R$ следует $(b, a) \in R$;
- антисимметричным, если для любых элементов $a, b \in A$ из $(a, b) \in R$ и $(b, a) \in R$ следует $a = b$.

Отношение эквивалентности (или просто «эквивалентность») – это произвольное рефлексивное, транзитивное и антисимметричное отношение. А отношение частичного порядка (или просто «частичный порядок» или даже «порядок») – это произвольное рефлексивное, транзитивное и антисимметричное отношение. Отношение предпорядка (или просто «предпорядок» или «квазипорядок») – это произвольное рефлексивное и транзитивное отношение.

брать» все $n!$ возможных перестановок элементов множества W . Но можно применить и так называемую топологическую сортировку, время работы которой пропорционально n^2 . Для всякого сортированного множества слов w_0, \dots, w_n мы будем предполагать существование константы $\text{first}: W \rightarrow W$, равной первому элементу w_0 , частичной операции $\text{next}: W \rightarrow W$, которая по произвольному элементу $w_i \in W$, $0 \leq i < n$, возвращает элемент $w_{(i+1)}$, и является неопределенной на w_n , а также частичной операции $\text{pre}: W \rightarrow 2^W$ такой, которая по произвольному элементу $w_i \in W$, $0 \leq i \leq n$ возвращает множество предыдущих элементов $w_0, \dots, w_{(i-1)}$, w_i .

Алгоритм синтаксического разбора Дж. Кока, Д. Янгера и Т. Касами.

- [
- 1) $G = (N, T, R, S)$ – грамматика в нормальной форме Хомского,
 - 2) α – непустое слово в терминальном алфавите T ,
 - 3) W – сортированное по включению множество подстрок α без пустого слова.
-] // Предусловие алгоритма.

VAR $\beta : W$;

VAR $db : 2^{W \times N}$;

$db := \emptyset$;

FOR ALL $\beta \in W$ IN ASCENDING ORDER DO

$db := db \cup \{ (\beta, n) : \beta \in T \ \& \ (n \rightarrow \beta) \in R \} \cup$

$\cup \{ (\beta, n) : \exists m', m'' \in N \ \& \ (n \rightarrow m' m'') \in R \ \&$

$\exists \gamma', \gamma'' (\beta \equiv \gamma' \gamma'' \ \& \ (\gamma', m') \in db \ \& \ (\gamma'', m'') \in db) \}$;

[

$\forall \beta \in W \ \forall n \in N$:

$(\beta, n) \in db \Leftrightarrow$

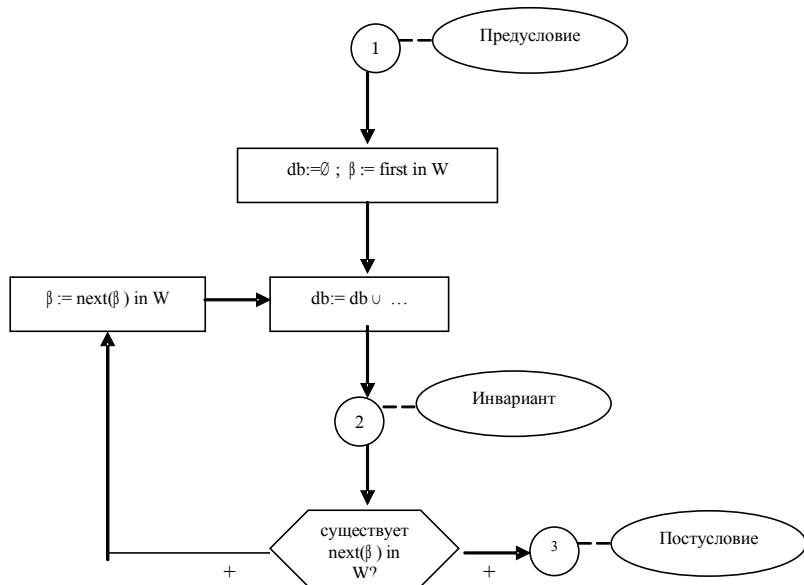
\Leftrightarrow существует дерево грамматического разбора слова β
в грамматике $G_n = (N, T, R, n)$

] // Постусловие алгоритма.

Теорема 3 (о корректности алгоритма Кока – Янгера – Касами) Алгоритм Дж. Кока, Д. Янгера и Т.Касами является тотально корректным.

Доказательство. Применим метод Флойда доказательства тотальной корректности детерминированных программ к описанному алгоритму, т. е. докажем его частичную корректность и завершаемость.

Заметим, что завершаемость алгоритма очевидна, поскольку он использует единственный цикл FOR по конечному множеству W . Поэтому остаётся доказать его частичную корректность. Для этого представим алгоритм в виде блок-схемы, выберем подходящие контрольные точки и снабдим их условиями корректности (см. рисунок ниже).



Предусловие и постусловие уже определены в описании алгоритма. В качестве инварианта контрольной точки 2 примем следующее:

W – сортированное множество подстрок α без пустого слова & $\beta \in W$ &

$\forall \beta' \in \text{pre}(\beta) \forall n \in \mathbb{N}$:

$(\beta', n) \in \text{db} \Leftrightarrow$

\Leftrightarrow существует дерево грамматического разбора слова β'
в грамматике $G_n = (N, T, R, n)$.

Разбор случаев (1 .. 2) и (2 – 3) представляет собой простое упражнение. Поэтому разберём только «основной» случай (2 + 2).

Пусть в контрольной точке (2) выполнен инвариант перед исполнением участка (2 + 2). Обозначим значения переменных β и db перед исполнением этого участка через β_0 и db_0 . Значит, имеет место

$$\beta_0 \in W \ \&$$

$$\forall \beta' \in \text{pre}(\beta_0) \ \forall n \in N:$$

$$(\beta', n) \in db_0 \Leftrightarrow$$

$$\Leftrightarrow \text{существует дерево грамматического разбора слова } \beta' \\ \text{в грамматике } G_n = (N, T, R, n).$$

Заметим, что сортированность W не изменяется во время выполнения тела цикла. Так как рассматривается случай (2 + 2), то условие «существует $\text{next}(\beta)$ in W » выполнено для β_0 , т. е. $\text{next}(\beta_0) \in W$. Обозначим его через $\beta_1 \in W$. Это значение β_1 есть значение переменной β после исполнения участка (2+2).

Значение db_1 переменной db после исполнения этого участка в соответствии с описанием алгоритма есть

$$db_1 := db_0 \cup \{ (\beta, n) : \beta \in T \ \& \ (n \rightarrow \beta) \in R \} \cup \\ \cup \{ (\beta, n) : \exists m', m'' \in N \ \& \ (n \rightarrow m'm'') \in R \ \& \\ \exists \gamma', \gamma'' (\beta \equiv \gamma'\gamma'' \ \& \ (\gamma', m') \in db_0 \ \& \ (\gamma'', m'') \in db_0) \}.$$

Поэтому для любых $\beta \subseteq \beta_1$ и $n \in N$ имеем:

$$(\beta, n) \in db_1 \Leftrightarrow$$

\Leftrightarrow верна одно из трех:

- $(\beta, n) \in db_0$;
- $\beta \in T$ и $(n \rightarrow \beta) \in R$;
- $\exists m', m'' \in N : (n \rightarrow m'm'') \in R \ \& \ \exists \gamma', \gamma'' (n \rightarrow m'm'') \in R \ \& \\ \& \ \exists \gamma', \gamma'' (\beta \equiv \gamma'\gamma'' \ \& \ (\gamma', m') \in db_0 \ \& \ (\gamma'', m'') \in db_0).$

В первом случае (когда $(\beta, n) \in db_0$) из факта, что инвариант имеет место для β_0 и db_0 , следует, что $(\beta, n) \in db_1$ равносильно существованию дерева грамматического разбора слова β' в грамматике $G_n = (N, T, R, n)$.

Во втором случае (когда $\beta \in T$ и $(n \rightarrow \beta) \in R$) получаем очевидным образом, что $(\beta, n) \in db_1$ равносильно существованию дерева грамматического разбора слова β в грамматике $G_n = (N, T, R, n)$ (которое, кстати, «состоит» из единственной продукции $(n \rightarrow \beta) \in R$).

В третьем случае надо использовать то, что инвариант имеет место для β_0 и db_0 . Пусть $m', m'' \in N$ и $\gamma', \gamma'' \subseteq \beta$ таковы, что $(n \rightarrow m'm'') \in R$, $\beta \equiv \gamma'\gamma''$, и $(\gamma', m'), (\gamma'', m'') \in db_0$. Так как грамматика находится в нормальной форме Хомского, то γ' и γ'' не пустые слова. Так как $\gamma' \subseteq \beta$ и $\gamma'' \subseteq \beta$, но ни γ' , ни γ'' не совпадают с β_1 , то $\gamma', \gamma'' \in \text{pre}(\beta_0)$. Поэтому имеем:

$$(\beta, n) \in db_1$$

$$\Downarrow \Uparrow$$

существует дерево грамматического разбора слова γ' в грамматике (N, T, R, m') ,

и

существует дерево грамматического разбора слова γ'' в грамматике $(N, T, R, m'') \Downarrow \Uparrow$

существует дерево грамматического разбора слова $\beta \equiv \gamma'\gamma''$ в грамматике (N, T, R, n) (которое, кстати, «начинается» с продукции $(n \rightarrow m'm'') \in R$).

Таким образом, мы показали, что инвариант точки (2) выполнен после исполнения участка $(2 + 2)$. Этим заканчивается доказательство частичной и тотальной корректности алгоритма. ■

«Стандартный» вариант алгоритма Кока – Янгера – Касами отличается от приведённого выше «абстрактного» варианта тремя моментами:

- вместо цикла по сортированному множеству подстрок используется два вложенных цикла; «внешний» цикл перебирает все возможные длины непустых подстрок (от 1 до $|\alpha|$), а «внутренний» – все варианты начальной позиции подстрок (от 0 до $|\alpha|$);
- все ограниченные кванторы реализуются переборными циклами типа FOR по перечислимым множествам;
- переменная db интерпретируется как матрица или реляционная база данных, т. е. трёхместное отношение, которое по любой подстроке β как по ключу «выдаёт» все такие $n \in N$, что $(\beta, n) \in db$.

В результате получается следующий

Алгоритм Дж. Кока, Д. Янгера и Т. Касами («стандартный»)

- [
- 1) $G = (N, T, R, S)$ – грамматика в нормальной форме,
 - 2) α – непустое слово в терминальном алфавите T ,
-] // Предусловие алгоритма.

VAR length, position, partition : $[1..|\alpha|]$;

VAR t: T; VAR n, m' , m'' : N;

VAR syntable : $[1..|\alpha|] \times [1..|\alpha|]$ array of 2^N ;

```

// Инициализация синтаксической таблицы:
FOR length :=1 TO  $|\alpha|$  DO
    FOR position :=1 TO ( $|\alpha| - \text{length} + 1$ ) DO
        syntable[position, (position + length)] :=  $\emptyset$  ;

// Заполнение синтаксической таблицы унарными productions:
FOR position :=1 TO  $|\alpha|$  DO // Перебор позиций.
    FOR EACH  $n \in N$  DO
        IF ( $n \rightarrow \alpha_{\text{position}}$ )  $\in R$ 
            THEN syntable[position, position] :=  $\{n\} \cup$ 
                syntable[position, position] ;

// Заполнение синтаксической таблицы бинарными productions:
FOR length :=1 TO  $|\alpha|$  DO
    // Перебор длин непустых подстрок.
    FOR position :=1 TO ( $|\alpha| - \text{length} + 1$ ) DO
        // Перебор начальных позиций непустых подстрок.
        FOR partition := (position + 1) TO (position + length) DO
            // Перебор разбиений непустых подстрок.
            FOR ALL  $n, m', m'' \in N$  DO
                IF ( $n \rightarrow m'm''$ )  $\in R$  &
                     $m' \in \text{syntable}[\text{position}, (\text{partition}-1)]$  &
                     $m'' \in \text{syntable}[\text{partition}, (\text{position} + \text{length}), m'']$ 
                THEN syntable[position, (position + length)] :=  $\{n\} \cup$ 
                    syntable[position, (position + length)] ;

[
 $\forall \text{position} \in [1..|\alpha|] \forall \text{length} \in (|\alpha| - \text{position} + 1) \forall n \in N:$ 
 $n \in \text{syntable}[\text{position}, (\text{position} + \text{length})] \Leftrightarrow$ 
     $\Leftrightarrow$  существует дерево грамматического разбора
        слова  $\alpha[\text{position} .. (\text{position} + \text{length})]$ 
        в грамматике  $G_n = (N, T, R, n)$ 
] // Постусловие алгоритма.

```

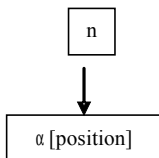
Теорема 4 (о верхней оценке сложности синтаксического анализа)
 Пусть G – фиксированная контекстно-свободная грамматика. Тогда для любого слова α в терминальном алфавите G дерево грамматического разбора α в грамматике G можно построить за время $O(|\alpha|^3)$.

Доказательство. Согласно теореме 2, не теряя общности, можно предполагать, что грамматика G находится в нормальной форме Н. Хомского. Поэтому для построения дерева грамматического разбора достаточно выполнить стандартный алгоритм Дж. Кока, Д. Янгера и Т. Касами, а потом «извлечь» дерево из построенной синтаксической таблицы. Используем, например, описанный выше стандартный вариант алгоритма, который строит syntable. Тогда рекурсивная процедура, возвращающая дерево синтаксического разбора подслова $\alpha[\text{position}..(\text{position}+\text{length})]$, где $1 \leq \text{position} \leq |\alpha|$ и $0 \leq \text{length} \leq (|\alpha| - \text{position} - 1)$, в грамматике G_n , где $n \in \mathbb{N}$, может быть задана следующим образом.

```

TREE(position, length);
IF syntable[position, length]  $\neq \emptyset$ 
THEN LET  $n \in \mathbb{N}$  – произвольный нетерминал из syntable[position,
length] IN
    IF length=0 THEN вернуть дерево

```



```

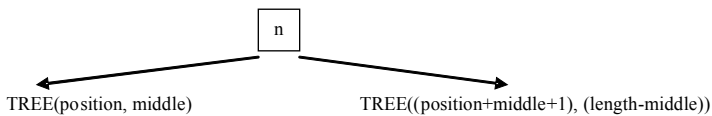
ELSE
BEGIN

```

```

    выбрать такое целое middle и пару нетерминалов  $m'$  и  $m''$ ,
    что
     $0 \leq \text{middle} \leq \text{length}$ ,  $m' \in \text{syntable}[\text{position}, \text{middle}]$ ,
     $m'' \in \text{syntable}[(\text{position} + \text{middle} + 1), (\text{length} - \text{middle})]$ ;
    и вернуть дерево

```



```

END

```

Кубическая оценка сложности следует из наличия в стандартном варианте алгоритма Дж. Кока, Д. Янгера и Т.Касами четырёх вложенных циклов FOR, причём число повторений каждого из трёх внешних ограничено

длиной слова α , а самый внутренний цикл FOR не зависит от размера слова α (а только от числа продукций в грамматике G). ■

Лемма о разрастании для контекстно-свободных языков

Материал предыдущей и данной лекции о регулярных и контекстно-свободных языках не является полным представлением математической теории этих языков, а только знакомит с основами теории лексического и синтаксического их анализа. (Подробнее см.: Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции.) Однако в заключении стоит привести ещё один факт общей теории.

Теорема 5 (лемма о разрастании для контекстно-свободных языков) Для всякого контекстно-свободного языка L существует такое число $k > 0$, что любое слово $\alpha \in L$, длина которого $|\alpha|$ больше k , можно представить в виде $\xi\beta\gamma\delta\theta$, так, что γ непустое слово, хотя бы одно из слов β и δ непустое слово, и для любого $i > 0$ слово $\xi\beta^i\gamma\delta^i\theta \in L$.

Доказательство. Пусть L – контекстно-свободный язык. Согласно теореме 2 можно считать, что L не содержит пустого слова и порождается некоторой грамматикой $G = (N, T, R, S)$ в нормальной форме Хомского. Пусть n – это число нетерминальных символов N , а $k = 2^n$. Пусть α произвольное слово языка $L(G)$, длина которого больше k .

Так как $\alpha \in L(G)$, то по теореме 1 это слово имеет дерево грамматического разбора TR в G . А так как G находится в нормальной форме Хомского, то TR – бинарное дерево. В силу того что количество листьев TR – это длина слова α , и $|\alpha| > k = 2^n$, то в TR есть обязательно путь длины более $n = \log_2 k$. Следовательно, на этом пути хотя бы один нетерминальный символ встретиться дважды. Пусть это символ q .

Следовательно, вывод слова α в G можно осуществить в несколько этапов:

- сначала вывести $S \Rightarrow \xi'q\theta'$, где $\xi'q\theta'$ – слово в TR на этаже, где встретилось первое вхождение q ;
- вывести $\xi' \Rightarrow \xi$ и $\theta' \Rightarrow \theta$, где ξ и θ – слова в терминальном алфавите в соответствии с TR ;
- вывести $q \Rightarrow \beta'q\delta'$, где $\beta'q\delta'$ – слово в поддереве TR , корнем которого является первое вхождение q , а β' и δ' не содержат q ;
- вывести $\beta' \Rightarrow \beta$ и $\delta' \Rightarrow \delta$, где β и δ – слова в терминальном алфавите в соответствии с TR ;
- и наконец, вывести $q \Rightarrow \gamma$ в соответствии с деревом TR .

Следовательно, α при таком выводе приобретает вид $\xi \beta \gamma \delta \theta$, причём γ непустое слово и хотя бы одно из слов β или δ непустое (в G нет продукций с пустой правой частью).

Теперь заметить, что если этапы 3 и 4 повторить ещё один раз, то получится вывод слова $\xi \beta^2 \gamma \delta^2 \theta$, если ещё два раза, то получится вывод слова $\xi \beta^3 \gamma \delta^3 \theta$, и так далее. ■

Лекция 9. Основы лексического и синтаксического анализа (на примере языка HeMo)

Формальный синтаксис языка HeMo

Формальный синтаксис большинства языков программирования состоит из двух частей: контекстно-свободного синтаксиса и контекстных зависимостей (ограничений). Так, например, для задания контекстно-свободного синтаксиса языка HeMo мы воспользуемся нотацией Бэкуса – Наура, а контекстные ограничения сформулируем полуформально (т. е. без использования какой-либо жёсткой нотации, например, атрибутивных грамматик).

Отметим, однако, что приведённая ниже формализация отнюдь не единственно возможная. Во-первых, потому, что неформальное определение языка HeMo слишком неоднозначно. А во-вторых, наша формализация ориентирована на человека, так как использует те же понятия, что и неформальное определение, не накладывает никаких дополнительных ограничений на формат правых частей определений и т. д. Такая формализация, однако, может оказаться неудобной для реализации алгоритмов построения дерева грамматического разбора, поскольку каждый алгоритм имеет свои специфические ограничения⁴⁶. Поэтому для реализации необходимо предварительно преобразовать человеко-ориентированную формализацию в эквивалентную, но машинно-ориентированную.

Альтернативой могло бы стать изначально машинно-ориентированное определение, но оно всё равно не единственно возможная формализация неформального варианта синтаксиса. Кроме того, чаще всего машинно-ориентированное определение бывает запутанным и неинтуитивным для человека, так как использует множество вспомогательных (искусственных) понятий, призванных подогнать формализацию под требуемый формат.

Определение. «Основу основ», ядро нашего варианта контекстно-свободного синтаксиса⁴⁷ языка HeMo составляют следующие шесть коротких определений:

⁴⁶ Например, алгоритм Кока – Янгера – Касами требует, чтобы для формализации использовалась контекстно-свободная грамматика в нормальной форме Хомского.

$\langle \text{программа} \rangle ::= \{ \langle \text{описание} \rangle ; \sim \} \langle \text{тело} \rangle$
 $\langle \text{описание} \rangle ::= \sim \text{VAR} \sim \langle \text{переменная} \rangle \sim \sim \langle \text{тип} \rangle$
 $\langle \text{тип} \rangle ::= \text{INT} \mid (\langle \text{тип} \rangle \sim \text{ARRAY} \sim \text{OF} \sim \langle \text{тип} \rangle)$
 $\langle \text{тело} \rangle ::= \langle \text{присваивание} \rangle \mid \langle \text{тест} \rangle \mid (\langle \text{тело} \rangle \sim ; \sim \langle \text{тело} \rangle) \mid ((\langle \text{тело} \rangle \sim \cup \sim \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle))$
 $*$
 $\langle \text{присваивание} \rangle ::= \langle \text{переменная} \rangle \sim := \sim \langle \text{выражение} \rangle$
 $\langle \text{тест} \rangle ::= \langle \text{отношение} \rangle \sim ?$

Далее следуют определения понятий, оставшихся неопределёнными в ядре:

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \mid$
 $\quad \langle \text{арифметическое_выражение} \rangle \mid$
 $\quad \langle \text{функциональное_выражение} \rangle$
 $\langle \text{операнд} \rangle ::= \langle \text{число} \rangle \mid \langle \text{переменная} \rangle$
 $\langle \text{арифметическое_выражение} \rangle ::= (\langle \text{операнд} \rangle \langle \text{знак_операции} \rangle \langle \text{операнд} \rangle)$
 $\langle \text{знак_операции} \rangle ::= + \mid - \mid \times \mid /$
 $\langle \text{функциональное_выражение} \rangle ::= \text{APP}(\langle \text{операнд} \rangle, \sim \langle \text{операнд} \rangle) \mid$
 $\quad \text{UPD}(\langle \text{операнд} \rangle, \sim \langle \text{операнд} \rangle, \sim \langle \text{операнд} \rangle)$
 $\langle \text{отношение} \rangle ::= \langle \text{операнд} \rangle \langle \text{символ_отношения} \rangle \langle \text{операнд} \rangle$
 $\langle \text{символ_отношения} \rangle ::= = \mid < \mid > \mid \leq \mid \geq$

Завершается контекстно-свободный синтаксис языка НеМо определениями понятий $\langle \text{число} \rangle$ и $\langle \text{переменная} \rangle$: переменная – это произвольная пос-

⁴⁷ В определении контекстно-свободного синтаксиса НеМо мы будем использовать метасимвол « \sim » для представления пробела. Таким образом, в определении все пробелы будут представлены явно посредством « \sim », неявных пробелов нет.

последовательность строчных латинских букв и арабских цифр, которая начинается с буквы, а число — это произвольная последовательность только из арабских цифр.

(Заметим, что наше определение сузило класс выражений, которые могут использоваться в присваиваниях, до выражений с одной операцией, сузило класс целочисленных выражений, которые могут использоваться в отношениях, до одночленов, а все тесты ограничило «тривиальными комбинациями» из одного отношения вместо пропозициональных комбинаций целочисленных отношений. Это пример неоднозначности неформального определения синтаксиса, которая может быть разрешена формальным определением, хотя, возможно, не самым удачным образом.)

Определение. Список контекстных зависимостей для формализованного варианта языка НеМо состоит всего из трёх дополнительных ограничений, которым должна удовлетворять всякая синтаксически правильная программа:

- 1) всякая встречающаяся переменная должна иметь единственное описание;
- 2) тип левой и правой части любого отношения должен быть INT;
- 3) типы левой и правой части любого присваивания должны совпадать.

Утверждение 1 Определённый выше диалект языка НеМо не является контекстно-свободным языком.

Доказательство. Предположим обратное, т. е. что наш диалект НеМо является контекстно-свободным языком. Тогда в соответствии с леммой о разрастании (теорема 5 предыдущей лекции 8) существует такое число $k > 0$, что любая НеМо-программа α , длина которой $|\alpha|$ больше k , можно представить в виде $\xi \beta \gamma \delta \theta$, так, что γ непустое слово, хотя бы одно из слов β и δ непустое слово, и для любого $i > 0$ слово $\xi \beta^i \gamma \delta^i \theta$ тоже является программой.

Возьмем в качестве α следующее слово

$$\sim \text{VAR} \sim \langle x^k \rangle \sim ; \sim \text{INT} ; \sim \langle x^k \rangle \sim := \sim \langle x^k \rangle,$$

где $\langle x^k \rangle$ представляет переменную, состоящую из k подряд идущих символов « x ». То, что это слово является программой, очевидно. То, что длина этой программы превосходит k , тоже очевидно. Следовательно, эту программу можно представить в виде $\xi \beta \gamma \delta \theta$ так, что γ непустое слово, хотя бы одно из слов β и δ непустое слово, и для любого $i > 0$ слово $\xi \beta^i \gamma \delta^i \theta$ тоже является программой. Теперь надо перебрать принципиально возможные значения β и δ и показать, что любой вариант приводит к противоречию.

Разберём, например, варианты β .

Если β это только самый первый пробел « \sim », то тогда $\xi \beta^i \gamma \delta^i \theta$ будет начинаться с нескольких пробелов, что для программы невозможно. Если β это не пробел « \sim », но некоторый префикс « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}$ », то тогда $\xi \beta^i \gamma \delta^i \theta$ будет начинаться с бессмысленного «заикания», например:

$\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT} \sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT} \sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT} \dots$

Варианты, когда β лежит в пределах от « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}; \sim$ » до « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}; \sim \langle x^k \rangle \sim \sim \langle x^k \rangle$ », также приводят к тому, что $\xi \beta^i \gamma \delta^i \theta$ будет начинаться с бессмысленного «заикания», например:

$\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}; \sim \langle x^k \rangle \sim \sim \text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}; \sim \langle x^k \rangle \sim \sim \dots$

Аналогично отменяются все варианты, когда β является всем описанием целиком « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT};$ » или частью одного из трёх вхождений $\langle x^k \rangle$ в программу.

В том случае, когда β – это всё описание целиком « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT};$ », слово $\xi \beta^i \gamma \delta^i \theta$ будет начинаться с многократного описания одной и той же переменной $\langle x^k \rangle$. А в случае, когда β является частью одного из трёх вхождений $\langle x^k \rangle$ в программу, соответствующее вхождение $\langle x^k \rangle$ заменится на $\langle x^{k+i \times m} \rangle$, а другие два вхождения $\langle x^k \rangle$ не изменятся, описанная и используемая переменные будут отличаться. Оба варианта для НеМо-программ запрещены контекстными ограничениями.

И так, мы перебрали все варианты β и показали, что они невозможны. Для δ варианты разбираются аналогично. Поэтому выбрать непустое значение β или γ в программе « $\sim\text{VAR} \sim \langle x^k \rangle \sim \sim\text{INT}; \sim \langle x^k \rangle \sim \sim \langle x^k \rangle$ » невозможно. Получается, что лемма о разрастании не применима к НеМо-программам. Следовательно, определённый нами диалект НеМо не является контекстно-свободным языком. ■

Задача лексического и синтаксического анализа

Определение 1. Задача синтаксического анализа для языка программирования, имеющего контекстно-свободный синтаксис и систему контекстных ограничений, состоит в том, что по входному тексту, созданному программистом и «претендующему» быть синтаксически правильной программой, необходимо:

- или построить дерево грамматического разбора этого текста, если он соответствует понятию «программа» контекстно-свободного синтаксиса,
- или отвергнуть этот текст и по возможности указать причины, т. е. синтаксические ошибки.

С теоретической точки зрения задача синтаксического анализа для любого языка программирования (имеющего контекстно-свободный синтаксис и систему контекстных ограничений) может быть решена следующим образом: достаточно преобразовать данное определение контекстно-свободного синтаксиса в контекстно-свободную грамматику в нормальной форме Хомского и применить алгоритм Кока – Янгера – Касами к входному тексту и этой грамматике. Однако такой подход в силу своей универсальности (т. е. применимости к любому контекстно-свободному языку) может оказаться неэффективным, поскольку не использует специфики конкретно языка (например, НеМо).

Разумеется, обычно особенности языка, которые могут повысить эффективность построения дерева грамматического разбора, выделяют явно в виде дополнительных формальных ограничений на грамматику языка, позволяющих вместо универсальных (и неэффективных) алгоритмов грамматического разбора использовать специализированные (эффективные) алгоритмы. Знакомство с примерами такого рода не входит в задачу данного курса, однако есть один общий приём, который оправдал себя на практике. Суть его состоит в выделении из задачи синтаксического анализа так называемой задачи лексического анализа.

Определение 2. Пусть дан какой-либо язык программирования с контекстно-свободным синтаксисом. Лексемами языка называются те его понятия, которые определяются регулярной грамматикой, а также совокупность (конечная) служебных слов и символов языка. Задача лексического анализа состоит в том, что во входном тексте, который «pretендует» быть программой, необходимо выделить все вхождения всех лексем этого языка (и, возможно, собрать сопутствующую информацию об этих лексемах).

В языке НеМо в частности есть лексемы {переменная} и {число}, а также зарезервированные слов и символы «ARRAY OF», «;», «{», «*», «(», «)».

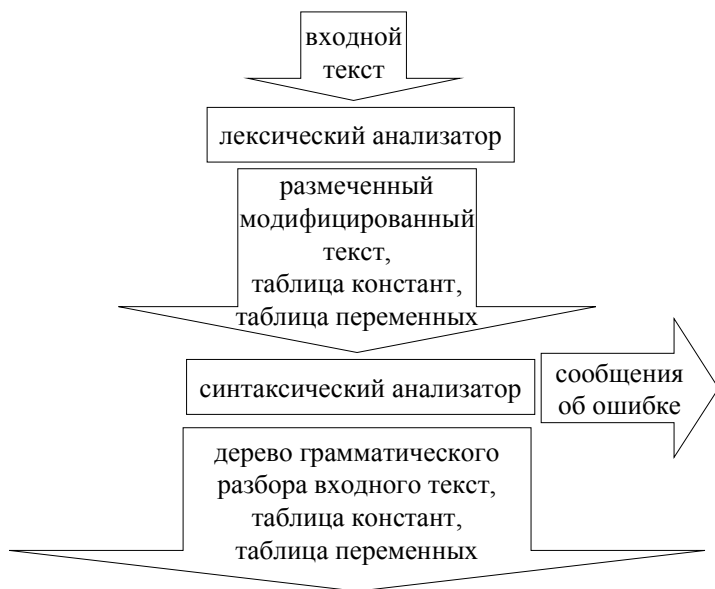
Отметим, что обычно лексический анализ выполним простым «сканированием входного текста» (т. е. однократным его просмотром слева направо), использующего конечные автоматы для распознавания лексем. Этот процесс называется «сканированием лексем».

Смысл выделения задачи лексического анализа из общей задачи синтаксического анализа многообразен. В частности, результаты лексического анализа могут применяться для более эффективного выполнения синтаксического анализа, так как расставляют своеобразные маяки (из выделенных и классифицированных лексем) во входном тексте, которые могут использоваться для облегчения распознавания грамматической структуры. В таком

случае лексический анализ превращается в особый этап синтаксического анализа, так сказать, его препроцессор. Эффективность указанного подхода проиллюстрируем на примере языка НеМо, формализованного описанным выше способом.

Вариант лексического и синтаксического анализатора для языка НеМо

Общая архитектура лексического и синтаксического анализатора для контекстно-свободного синтаксиса языка НеМо представлена на следующем рисунке, где прямоугольники обозначают функциональные модули, а стрелки – потоки управления и данных.



Оба представленных модуля используют входной текст, содержащий «программу» и который можно читать и модифицировать посимвольно, а также две дополнительные структуры данных TV и ТС типа множество:

- TV предназначена для «учётных записей» (⟨переменная⟩, псевдоним, список типов ⟨переменной⟩) о встреченных переменных, о внутреннем представлении их имён (их псевдонимах) и типах этих переменных;

- ТС предназначена для «учётных записей» ($\langle \text{число} \rangle$, псевдоним) о встреченных числах (константах) и об их внутреннем представлении (их псевдонимах).

Эти две структуры мы будем называть таблицами переменных и констант соответственно. Псевдонимы, использующиеся в этих таблицах, это уникальные символы, которые порождаются при помощи функции NEW(псевдоним).

Модуль NeMoLeX лексического анализа работает следующим образом. (Алгоритм лексического анализа NeMoLeX для языка HeMo.)

[Входной текст записан с использованием пробелов, заглавных и строчных латинских букв, арабских цифр и дополнительных символов «:», «=», «<», «>», «≤», «≥», «+», «-», «*», «/», «;», «⊂», «(», «)».] // Предусловие модуля.

1. Открыть файл входного текста для чтения начиная с самого левого символа. Инициализировать счётчик глубины скобок $P := 0$, сформировать пустые таблицы переменных и констант $TV := \emptyset$ и $TC := \emptyset$.
2. Пока не достигнут конец файла текста, выполнять цикл 2.1–2.7:
 - 2.1) если обозреваемый символ – строчная латинская буква, то сканировать посимвольно $\langle \text{переменную} \rangle$ по файлу текста вплоть до первой небуквы и нецифры (не включая её);
 - 2.1.1) если сканированная $\langle \text{переменная} \rangle$ не имеет учётной записи в таблице переменных TV , то добавить учётную запись ($\langle \text{переменная} \rangle$, NEW(псевдоним), пустой список типов) в таблицу переменных $TV := TV \cup \{(\langle \text{переменная} \rangle, \text{NEW(псевдоним)}, \text{пустой список типов})\}$;
 - 2.1.2) извлечь псевдоним сканированной $\langle \text{переменной} \rangle$ из таблицы TV , подставить его вместо $\langle \text{переменной} \rangle$ в файл текста и перейти на шаг 2.7;
 - 2.2) если обозреваемый символ – арабская цифра, то сканировать посимвольно $\langle \text{число} \rangle$ по файлу текста вплоть до первой нецифры (не включая её);
 - 2.2.1) если сканированное $\langle \text{число} \rangle$ не имеет учётной записи в таблице констант TC , то добавить учётную запись ($\langle \text{число} \rangle$, NEW(псевдоним)) в таблицу констант $TC := TC \cup \{(\langle \text{число} \rangle, \text{NEW(псевдоним)})\}$;

- 2.2.2) извлечь псевдоним сканированного ⟨числа⟩ из таблицы ТС, подставить его вместо ⟨числа⟩ в файл текста и перейти на шаг 2.7;
 - 2.3) если обозреваемый символ «(», то увеличить счётчик глубины $P := P + 1$, пометить эту скобку в файле текста тэгом, равным текущему значению P , и перейти на шаг 2.7;
 - 2.4) если обозреваемый символ «)», то пометить эту скобку в файле текста тэгом, равным текущему значению P , уменьшить счётчик глубины $P := P - 1$ и перейти на шаг 2.7;
 - 2.5) если обозреваемый символ «;» или «⌋», то пометить этот символ в файле текста тэгом, равным текущему значению P , и перейти на шаг 2.7;
 - 2.6) если последние сканированные символы, включая обозреваемый символ, образуют «ARRAY OF», то пометить это слово в файле текста тэгом, равным текущему значению P , и перейти на шаг 2.7;
 - 2.7) перейти на следующий справа символ входного текста.
 3. Закрыть файл текста и выдать его совместно с таблицами TV и ТС в качестве результатов.
- [
1. Таблицы TV и ТС содержат учётные записи для тех и только тех ⟨переменных⟩ и ⟨чисел⟩ соответственно, которые встречаются во входном тексте, причём каждая такая ⟨переменная⟩ и ⟨число⟩ имеют уникальную учётную запись с уникальным псевдонимом, а список типов ⟨переменной⟩ пуст.
 2. Выходной текст записан с использованием пробелов, заглавных и строчных латинских букв, арабских цифр, дополнительных символов «:», «=», «<», «>», «≤», «≥», «+», «-», «*», «/», «;», «⌋», «(», «)» и псевдонимов, причём, каждое вхождение «ARRAY OF», «;», «⌋», «(», «)» снабжено тэгом, значение которого равно глубине скобочной вложенности этого вхождения, при этом, однако, выходной текст не содержит ни ⟨переменных⟩, ни ⟨чисел⟩, а входной текст получается из выходного в результате подстановки вместо всех псевдонимов соответствующих ⟨переменных⟩ и ⟨чисел⟩, которые соответствуют этим псевдонимам по таблице переменных и констант TV и ТС.
-] // Постусловие модуля.

Модуль NeMoSyN синтаксического анализа вызывает две рекурсивные процедуры DECL и BODY, а процедура DECL вызывает в свою очередь процедуру TYPE. Этот модуль и все эти процедуры используют две

глобальные переменные TV и TC, в которых хранятся таблицы переменных и констант, а также параметр-значение текстового типа.

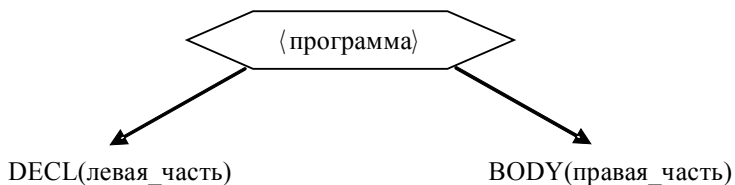
Алгоритм синтаксического анализа NeMoSyN для языка HeMo.

[

1. Таблицы TV и TC содержат учётные записи для тех и только тех ⟨переменных⟩ и ⟨чисел⟩ соответственно, псевдонимы которых встречаются во входном тексте, причём каждая такая ⟨переменная⟩ и ⟨число⟩ имеют уникальную учётную запись с уникальным псевдонимом, а список типов ⟨переменной⟩ пуст.
2. Входной текст записан с использованием пробелов, заглавных и строчных латинских букв, арабских цифр, дополнительных символов «:», «=», «<», «>», «≤», «≥», «+», «-», «*», «/», «;», «⋈», «(», «)» и псевдонимов, причём каждое вхождение «ARRAY OF», «;», «⋈», «(», «)» снабжено тэгом, значение которого равно глубине скобочной вложенности этого вхождения, при этом, однако, входной текст не содержит ни ⟨переменных⟩, ни ⟨чисел⟩.

] // Предусловие модуля.

1. Открыть файл входного текста для чтения начиная с самого левого символа.
2. Если в файле текста есть вхождение пары символов «;~», в которой «;» помечен тегом скобочной вложенности 0,
 - 2.1) то выделить в файле текста самое правое такое вхождение, и пусть левая_часть – это префикс текста вплоть до и включая выделенное вхождение «;~», а правая_часть – это суффикс текста после выделенного вхождения «;~»,
 - 2.2) иначе пусть левая_часть – это пустое слово, а правая – весь текст.
3. Вызвать процедуры DECL(левая_часть) и BODY(правая_часть).
4. Если обе процедуры не сообщили об ошибке, а построили соответствующие деревья
 - 4.1) то сформировать дерево



- 4.2) иначе сообщить об ошибках, обнаруженных этими процедурами.

5. Закрывать файл текста.

[

1. Таблицы TV и TC содержат учётные записи для тех и только тех <переменных> и <чисел> соответственно, псевдонимы которых встречаются во входном тексте, причём каждая такая <переменная> и <число> имеют уникальную учётную запись с уникальным псевдонимом, а список типов <переменной> содержит те и только те типы, с которыми псевдоним этой переменной описан во входном тексте.
2. Если после подстановки во входной текст вместо псевдонимов переменных и чисел, соответствующих им согласно таблицам переменных и констант TV и TC, получается <программа>,
 - 2.1) то выходом является дерево её грамматического разбора,
 - 2.2) иначе выходом является сообщение об ошибке.

] // Постусловие модуля.

Инвариантом всех процедур DECL, BODY и TYPE, который входит под номером один во все их пред- и постусловия, является следующее общее условие: таблицы TV и TC содержат учётные записи для тех и только тех <переменных> и <чисел> соответственно, псевдонимы которых встречаются во входном тексте, причём каждая такая <переменная> и <число> имеют уникальную учётную запись с уникальным псевдонимом. Поэтому мы не будем воспроизводить этот инвариант явно, а будем приводить только специфические условия.

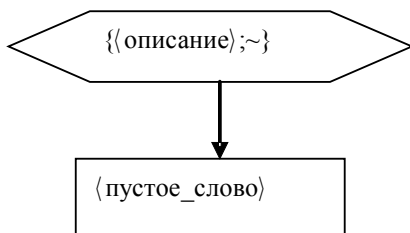
Процедура DECL(текст)

[

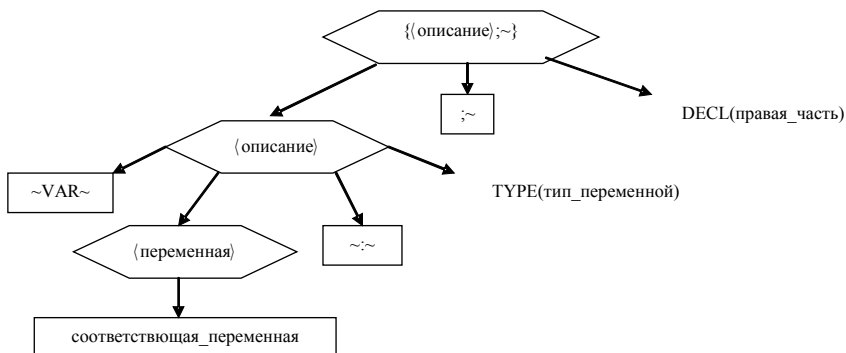
1. (См. общее условие выше.)
2. Параметр «текст» является фрагментом входного текста модуля NeMoSyN.

] // Предусловие процедуры.

1. Если текст пуст, то сформировать следующее дерево и завершить работу:



2. Если текст не пуст, то
 - 2.1) если текст не содержит пары символов «;~»
 - 2.1.1) выдать сообщение об ошибке «описание должно заканчиваться ;~» и завершить работу,
 - 2.1.2) иначе выделить в тексте самое левое вхождение «;~»;
 - 2.2) пусть левая_часть – это префикс текста вплоть до и включая выделенное вхождение «;~», а правая_часть – это суффикс текста начиная сразу после выделенного вхождения «;~»;
 - 2.3) если левая_часть не начинается с «~VAR~псевдоним_переменной~:~», то сообщить об ошибке «текст не может быть описанием переменной» и завершить работу;
 - 2.4) если левая_часть начинается с «~VAR~псевдоним_переменной~:~», то пусть соответствующая_переменная – это ⟨переменная⟩, которая соответствует псевдониму_переменной по таблице переменных TV, а тип_переменной – это часть левой_части после «~VAR~псевдоним_переменной~:~» до «;~».
3. Вызвать процедуры TYPE(тип_переменной) и DECL(правая_часть).
4. Если обе процедуры не сообщили об ошибке, а построили деревья
 - 4.1) то сформировать следующее дерево и добавить в список типов соответствующей_переменной в таблице переменных тип_переменной

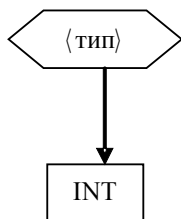


- 4.2) иначе сообщить об ошибках, обнаруженных этими процедурами.
- [
1. (См. общее условие выше.)

2. Если после подстановки в текст вместо псевдонимов переменных, которые соответствуют им согласно таблицам переменных и констант TV и TC, получается {⟨описание;~⟩},
 - 2.1) то выходом является дерево грамматического разбора этого текста в соответствии с определением {⟨описание;~⟩}, таблица констант TC не меняется, а таблица переменных TV отличается от её значения перед выполнением процедуры только тем, что в учётной записи соответствующей_переменной к списку типов добавлен тип_переменной;
 - 2.2) иначе выходом является сообщение об ошибке.
-] \\ Постусловие модуля.

Процедура TYPE(текст)

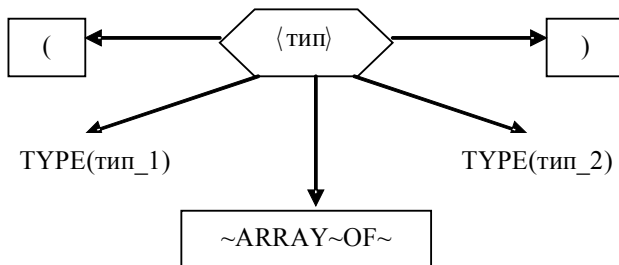
- [
1. (См. общее условие выше.)
 2. Параметр «текст» является фрагментом входного текста модуля NeMoSyN.
-] // Предусловие процедуры.
1. Если текст пуст, то сообщить об ошибке «тип не может быть пустым» и завершить работу.
 2. Если текст – это «INT», то сформировать следующее дерево и завершить работу:



3. Если текст не пуст и не «INT», то
 - 3.1) если текст не содержит символов «(» и «)», то выдать сообщение об ошибке «текст не может быть типом» и завершить работу;
 - 3.2) выделить в тексте левое вхождение символа «(» и выделить в тексте правое вхождение символа «)»; пусть P – значение тэга выделенного символа «)»;
 - 3.3) пусть левая_часть – это префикс текста до (не включая) выделенного вхождения «(», а правая_часть – это суффикс текста начиная сразу после (не включая) выделенного вхождения «)», корень –

это часть текста между выделенными вхождениями «(» и «)» (не включая их);

- 3.4) если левая_часть или правая_часть непусты, то сообщить об ошибке «текст не может быть типом переменной» и завершить работу;
- 3.5) если левая_часть и правая_часть пусты, то проверить, существует ли внутри корня «~ARRAY~OF~», также имеющее значение тэга Р;
- 3.6) если такого вхождения не существует,
 - 3.6.1) выдать сообщение об ошибке «текст не может быть типом» и завершить работу,
 - 3.6.2) иначе выделить внутри корня левое вхождение у которого «~ARRAY~OF~» также имеет значение тэга Р,
 - 3.6.3) пусть тип_1 – это префикс корня вплоть до (не включая) выделенного вхождения «~ARRAY~OF~»,
 - 3.6.4) тип_2 – это суффикс корня после (не включая) выделенного вхождения «~ARRAY~OF~».
4. Вызвать процедуры TYPE(тип_1) и TYPE(тип_2).
5. Если обе процедуры не сообщили об ошибке, а построили деревья
 - 5.1) то сформировать дерево



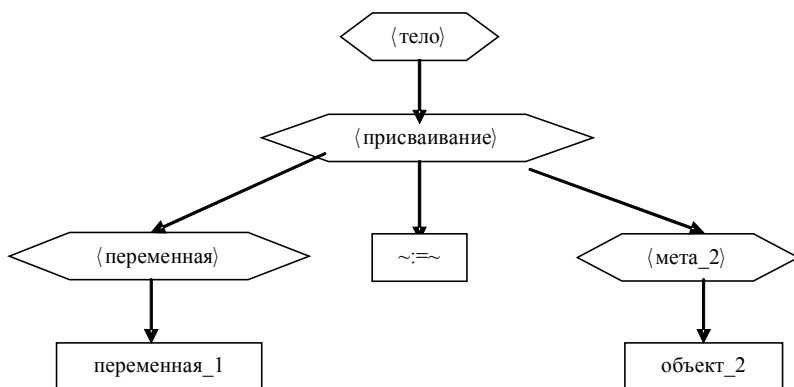
5.2) иначе сообщить об ошибках, обнаруженных этими процедурами.

[

1. (См. общее условие выше.)
 2. Если текст является <типом>,
 - 2.1) то выходом является дерево грамматического разбора этого текста в соответствии с определением <тип>,
 - 2.2) иначе – выходом является сообщение об ошибке.
-] \ \ Постусловие процедуры.

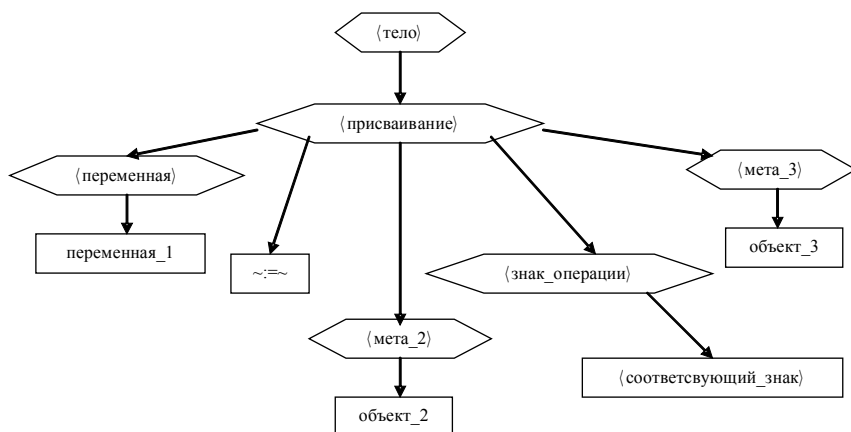
Процедура BODY(текст)

- [
1. (См. общее условие выше.)
 2. Параметр «текст» является фрагментом входного текста модуля NeMoSyN.
-] // Предусловие процедуры.
1. Если текст пуст, то сообщить об ошибке «тело программы не может быть пустым» и завершить работу.
 2. Если текст – это «псевдоним_1~::~~псевдоним_2», причём псевдоним_1 – это псевдоним переменной, то
 - 2.1) пусть
 - 2.1.1) переменная_1 – это ⟨переменная⟩, соответствующая псевдониму_1 по таблице переменных TV,
 - 2.1.2) объект_2 – это ⟨переменная⟩ или ⟨число⟩, соответствующий псевдониму_2 по таблицам переменных TV или констант ТС,
 - 2.1.3) а мета_2 – это соответственно метапонятие ⟨переменная⟩ или метапонятие ⟨число⟩;
 - 2.2) сформировать следующее дерево и завершить работу:

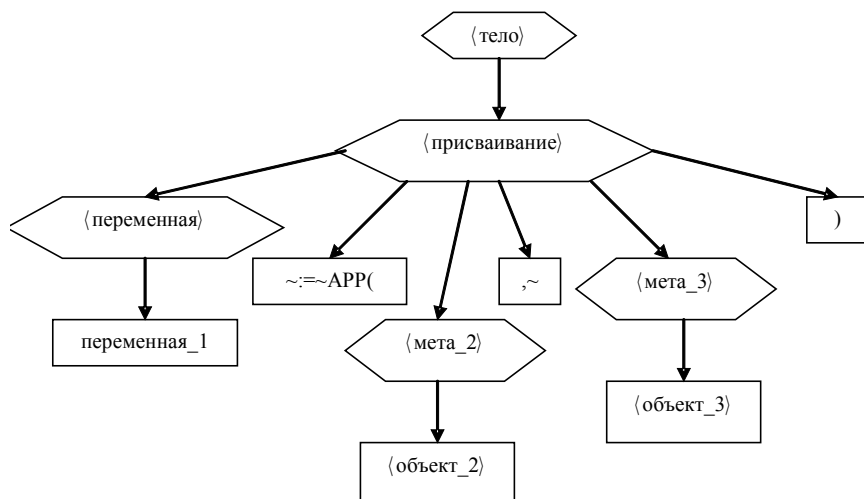


3. Если текст – это
 - 3.1) или «псевдоним_1~::~~псевдоним_2 + псевдоним_3»,
 - 3.2) или «псевдоним_1~::~~псевдоним_2 * псевдоним_3»,
 - 3.3) или «псевдоним_1~::~~псевдоним_2 – псевдоним_3»,
 - 3.4) или «псевдоним_1~::~~псевдоним_2/псевдоним_3»,
 - причём псевдоним_1 – это псевдоним переменной, то
 - 3.5) пусть

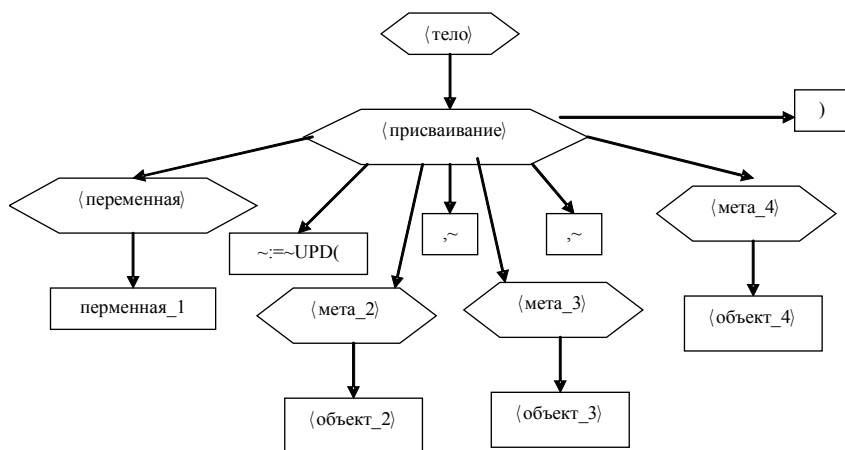
- 3.5.1) переменная_1 – это \langle переменная \rangle , соответствующая псевдониму_1 по таблице переменных TV,
- 3.5.2) объект_2 и объект_3 – это \langle переменная \rangle или \langle число \rangle , соответствующие псевдониму_2 и псевдониму_3 по таблицам переменных TV или констант ТС,
- 3.5.3) мета_2 и мета_3 – это метапонятие \langle переменная \rangle или метапонятие \langle число \rangle соответственно,
- 3.5.4) а соответствующий_знак – это «+», «*», «-» или «/»;
- 3.6) сформировать следующее дерево и завершить работу:



4. Если текст – это «псевдоним_1~::~~APP(псевдоним_2,~псевдоним_3)», причём псевдоним_1 – это псевдоним переменной, то пусть
- 4.1) переменная_1 – это \langle переменная \rangle , соответствующая псевдониму_1 по таблице переменных TV,
- 4.2) объект_2 и объект_3 – это \langle переменная \rangle или \langle число \rangle , соответствующие псевдониму_2 и псевдониму_3 по таблицам переменных TV или констант ТС,
- 4.3) мета_2 и мета_3 – это метапонятие \langle переменная \rangle или метапонятие \langle число \rangle соответственно;
- 4.4) сформировать следующее дерево и завершить работу:



5. Если текст – это «псевдоним_1~:=~UPD(псевдоним_2,~псевдоним_3,~псевдоним_4)», причём псевдоним_1 – это псевдоним переменной, то пусть
 - 5.1) переменная_1 – это <переменная>, соответствующая псевдониму_1 по таблице переменных TV,
 - 5.2) объект_2, объект_3 и объект_4 – это <переменная> или <число>, соответствующие псевдониму_2, псевдониму_3 и псевдониму_4 по таблицам переменных TV или констант ТС,
 - 5.3) мета_2, мета_3 и мета_4 – это метапонятие <переменная> или метапонятие <число> соответственно;
 - 5.4) сформировать следующее дерево и завершить работу:



6. Если текст – это

6.1) или «псевдоним_1 = псевдоним_2~?»,

6.2) или «псевдоним_1 < псевдоним_2~?»,

6.3) или «псевдоним_1 ≤ псевдоним_2~?»,

6.4) или «псевдоним_1 > псевдоним_2~?»,

6.5) или «псевдоним_1 ≥ псевдоним_2~?»,

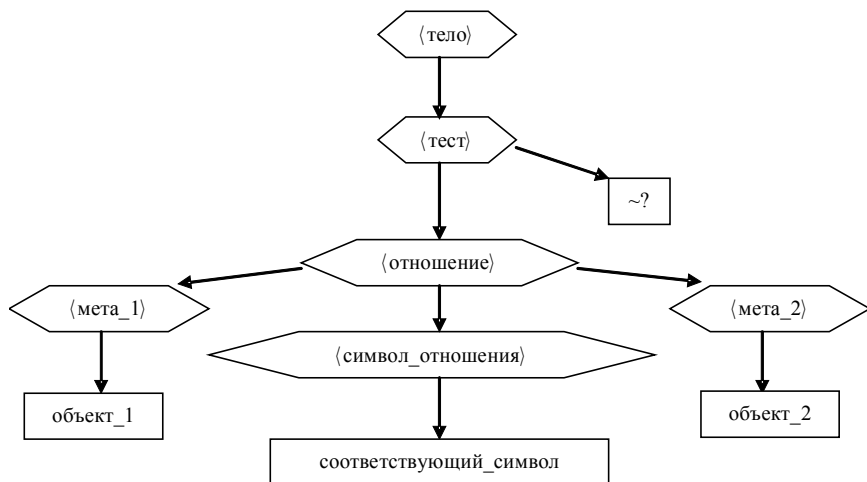
6.6) то пусть

6.6.1) объект_1 и объект_2 – это <переменная> или <число>, соответствующие псевдониму_1 и псевдониму_2 по таблицам переменных TV или констант ТС,

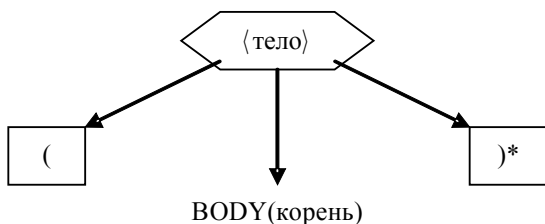
6.6.2) мета_1 и мета_2 – это <переменная> или <число> соответственно,

6.6.3) а соответствующий_символ – это «=», «<», «≤», «>» или «≥»;

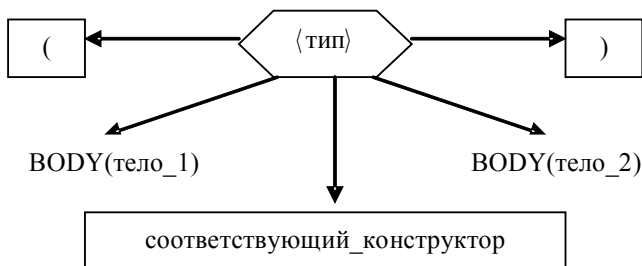
6.7) сформировать следующее дерево и завершить работу:



7. Если текст не удовлетворяет условиям предыдущих пунктов, то
 - 7.1) если текст не содержит символов «(» и «)», выдать сообщение об ошибке «текст не может быть телом программы» и завершить работу;
 - 7.2) выделить в тексте левое вхождение символа «(» и выделить в тексте правое вхождение символа «)»; пусть P – значение тэга выделенного символа «(»;
 - 7.3) пусть левая_часть – это префикс текста до (не включая) выделенного вхождения «(», а правая_часть – это суффикс текста начиная сразу после (не включая) выделенного вхождения «)», а корень – это часть текста между выделенными вхождениями «(» и «)» (не включая их);
 - 7.4) если левая_часть не пуста, то сообщить об ошибке «текст не может быть телом программы» и завершить работу;
 - 7.5) если правая_часть не пуста и не «*», то сообщить об ошибке «текст не может быть телом программы» и завершить работу;
 - 7.6) если правая часть «*», то
 - 7.6.1) вызвать процедуру BODY(корень),
 - 7.6.2) если процедура не сообщила об ошибке, а построила дерево,
 - 7.6.3) то сформировать следующее дерево и завершить работу:



- 7.6.4) иначе сообщить об ошибках, обнаруженных этой процедурой, и завершить работу;
- 7.7) если правая_часть пуста, то проверить, существует ли внутри корня «~;~» или «~∪~», которое также имеет значение тэга Р;
- 7.8) если такого вхождения не существует,
- 7.8.1) выдать сообщение об ошибке «текст не может быть телом программы» и завершить работу,
- 7.8.2) иначе выделить внутри корня левое вхождение у которого «~;~» или «~∪~» также имеет значение тэга Р;
- 7.8.3) пусть соответствующий_конструктор это выделенное вхождение,
- 7.8.4) пусть тело_1 – это префикс корня вплоть до (не включая) выделенного вхождения,
- 7.8.5) тело_2 – это суффикс корня после (не включая) выделенного вхождения.
8. Вызвать процедуры BODY(тело_1) и BODY(тело_2).
9. Если обе процедуры не сообщили об ошибке, а построили деревья,
- 9.1) то сформировать дерево



- 9.2) иначе сообщить об ошибках, обнаруженных этими процедурами.
- [

1. (См. Общее условие выше.)
 2. Если текст является $\langle \text{телом} \rangle$,
 - 2.1) то выходом является дерево грамматического разбора этого текста в соответствии с определением $\langle \text{тело} \rangle$,
 - 2.2) иначе выходом является сообщение об ошибке.
-] // Постусловие модуля.

Лекция 10. Корректность и сложность синтаксического анализа (на примере языка HeMo)

Корректность синтаксического анализа для HeMo

Первое отличие, которое сразу бросается в глаза при сравнении описанного выше лексического и синтаксического анализаторов от алгоритма Дж. Кока, Д. Янгера и Т. Касами, состоит в том, что общий алгоритм является восходящим, а наш специализированный алгоритм является нисходящим:

- классический алгоритм сначала строит деревья грамматического разбора для односимвольных фрагментов входного текста, потом для фрагментов из двух символов, затем для фрагментов из трёх символов и т. д., пока не дойдёт до «фрагмента», состоящего из всего входного текста;
- наш алгоритм, наоборот, начинает строить дерево грамматического разбора со всего входного текста, разбить его на фрагменты, для которых или строит дерево разбора явно, или рекурсивно применяет эту же процедуру.

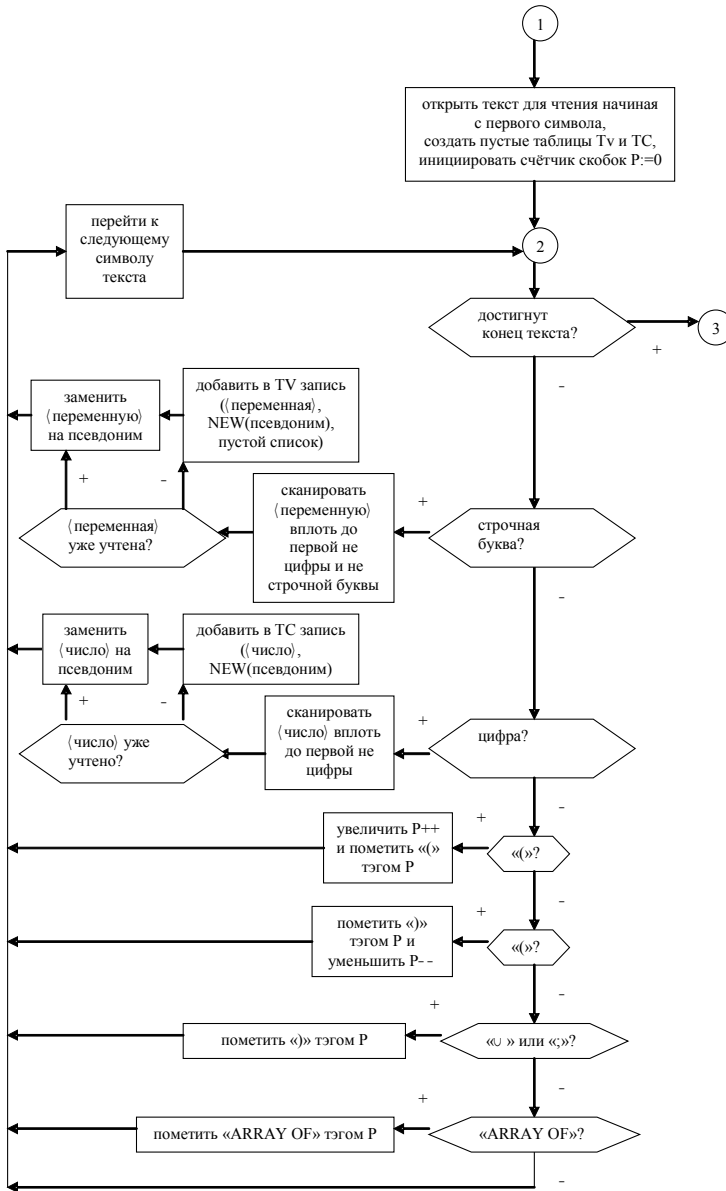
Описанный нами в предыдущей лекции 9 вариант лексического и синтаксического разбора для HeMo является примером так называемого нисходящего леворекурсивного разбора. Мы не будем изучать общую теорию, относящуюся к данному методу, поскольку это уведёт нас за пределы курса, однако докажем корректность представленного алгоритма.

Утверждение 1 Алгоритмы NeMoLeX, TYPE, DECL, BODY и NeMoSyN являются totally корректными по отношению к своим пред- и постусловиям.

Доказательство проведём в порядке NeMoLeX, TYPE, BODY, DECL и NeMoSyN. Доказательство для NeMoLeX проведём методом Р. Флойда, доказательство для TYPE и BODY выполним индукцией по длине входного текста, а доказательства для DECL и NeMoSyN получим, комбинируя доказательства для TYPE и BODY.

Тотальная корректность модуля NeMoLeX. Метод Флойда доказательства тотальной корректности состоит в доказательстве частичной корректности и доказательстве завершаемости. Доказательство завершаемости состоит в том, что при каждой итерации тела единственного цикла (пункты 2.1 – 2.7) обязательно выполняется шаг 2.7, который уменьшает длину непрочитанной части входного текста. Для доказательства частичной корректности представим модуль NeMoLeX в виде блок-схемы (см. рисунок на следующей странице), и сопоставим точкам 1 и 3 на блок-схеме (входу и выходу модуля) его пред- и постусловия соответственно, а точке 2 (через которую проходит любой цикл в блок-схеме) – сопоставим конъюнкцию следующих трех условий:

1.
 - 1.1) обозреваемый символ текста не является частью переменной или числа, т. е. если этот символ строчная буква или цифра, то его непосредственный предшественник не является ни строчной буквой, ни цифрой;
 - 1.2) значение счётчика Р равно глубине скобочной вложенности обозреваемого символа текста.
2. Таблицы TV и TC содержат учётные записи для тех и только тех *⟨переменных⟩* и *⟨чисел⟩* соответственно, которые встречаются в *прочитанной части* входного текста, причём каждая такая *⟨переменная⟩* и *⟨число⟩* имеют уникальную учётную запись с уникальным псевдонимом, а список типов *⟨переменной⟩* пуст.
3. Текст записан с использованием пробелов, заглавных и строчных латинских букв, арабских цифр, дополнительных символов «:», «=», «<», «>», «≤», «≥», «+», «-», «*», «/», «;», «⋈», «(», «)» и псевдонимов, причём
 - 3.1) каждое вхождение «ARRAY OF», «;», «⋈», «(», «)» в *прочитанной части* текста снабжено тэгом, значение которого равно глубине скобочной вложенности этого вхождения,
 - 3.2) *прочитанная часть* текста не содержит ни *⟨переменных⟩*, ни *⟨чисел⟩*, а входной текст получается из выходного текста в результате подстановки вместо всех псевдонимов соответствующих *⟨переменных⟩* и *⟨чисел⟩*, которые соответствуют этим псевдонимам по таблице переменных и констант TV и TC.



Теперь остаётся только показать, что для каждого ациклического участка если условие, сопоставленное началу участка, верно перед началом исполнения участка, то условие, сопоставленное концу участка, верно после исполнения этого участка. Для участка (1..2) это тривиально, так как после открытия текста для чтения и инициализации таблиц и счётчика прочитанная часть текста пуста. Для участка (2+3) дело обстоит так же просто, ибо в таком случае прочитанная часть – это весь текст. Остаётся рассмотреть семь различных ациклических участков (2 – + 2), (2 – – + 2), (2 – – – + 2), (2 – – – – + 2), (2 – – – – – + 2), (2 – – – – – + 2) и (2 – – – – – – 2). Это трудоёмкое, но простое по смыслу упражнение остаётся для самостоятельной работы.

Тотальная корректность процедур TYPE и BODY доказывается аналогично друг другу индукцией по длине текста, поэтому мы разберём только доказательство для TYPE, а доказательство для BODY остаётся для самостоятельной работы.

База индукции: пустой текст. Пустой текст не является типом в силу определения $\langle \text{тип} \rangle ::= \text{INT} \mid \langle \text{тип} \rangle \sim \text{ARRAY} \sim \text{OF} \sim \langle \text{тип} \rangle$. В то же время шаг 1 процедуры TYPE для пустого текста предписывает выдать сообщение об ошибке и на том завершить работу. Тем самым доказательство базы индукции полностью исчерпано.

Индукционная гипотеза: предположим, что для всякого текста, длина которого не больше некоторого $m \geq 0$, если выполнены предусловия процедуры TYPE, то эта процедура завершает свою работу над текстом с результатом, удовлетворяющим постусловиям процедуры.

Шаг индукции. Пусть текст состоит из $n = (m + 1)$ символов и удовлетворены предусловия процедуры TYPE. Так как текст непуст, то шаг 1 процедуры неприменим. В силу определения имеем:

$$\langle \text{тип} \rangle ::= \text{INT} \mid (\langle \text{тип} \rangle \sim \text{ARRAY} \sim \text{OF} \sim \langle \text{тип} \rangle).$$

Если текст – это INT, то (согласно шагу 2 процедуры TYPE) данная процедура построит дерево грамматического разбора этого текста и завершит работу.

Если же текст – это не INT, то (согласно определению типа) он является типом тогда и только тогда, когда этот текст может быть представлен в виде

$$(\text{тип_1 ARRAY OF тип_2}),$$

где тип_1 и тип_2 – тоже типы. Значит, текст является типом тогда и только тогда, когда он начинается со скобки «(», заканчивается скобкой «)», содержит «~ARRAY~ OF~» со скобочной вложенностью такой же, как у начина-

ющей текст «(», и (согласно индукционной гипотезе для тип_1 и тип_2) оба вызова TYPE(тип_1) и TYPE(тип_2) завершаются построением деревьев грамматического разбора для тип_1 и тип_2 соответственно. Но это в точности то, что происходит на шагах 3, 4 и 5 процедуры TYPE(текст). Следовательно, и в этом случае процедура TYPE(текст) завершается построением дерева грамматического разбора тогда и только тогда, когда текст – это тип.

Таким образом, доказательство тотальной корректности процедуры TYPE завершено.

Тотальная корректность процедуры DECL доказывается также индукцией по длине текста с использованием уже доказанной тотальной корректности процедуры TYPE.

База индукции: пустой текст. Пустой текст подпадает под определение $\{\langle \text{описание} \rangle; \sim\}$. В то же время шаг 1 процедуры DECL для пустого текста предписывает построить соответствующее дерево и на том завершить работу. Тем самым доказательство базы индукции полностью исчерпано.

Индукционная гипотеза: предположим, что для всякого текста, длина которого не больше некоторого $m \geq 0$, если выполнены предусловия процедуры DECL, то эта процедура завершает свою работу над текстом с результатом, удовлетворяющим постусловиям процедуры.

Шаг индукции. Пусть текст состоит из $n = (m + 1)$ символов и удовлетворены предусловия процедуры DECL. Так как текст не пуст, то шаг 1 процедуры неприменим. В силу определения метаскобок $\{ \}$ имеем:

$\{\langle \text{описание} \rangle; \sim\} = \text{пустое_слово} \mid \langle \text{описание} \rangle; \sim \{\langle \text{описание} \rangle; \sim\}$.

Согласно этому соотношению и определению

$\langle \text{описание} \rangle ::= \sim \text{VAR} \sim \langle \text{переменная} \rangle \sim : \sim \langle \text{тип} \rangle$

имеем

$\{\langle \text{описание} \rangle; \sim\} = \text{пустое_слово} \mid \sim \text{VAR} \sim \langle \text{переменная} \rangle \sim : \sim \langle \text{тип} \rangle; \sim \{\langle \text{описание} \rangle; \sim\}$.

Значит, непустой текст с подставленными переменными и числами вместо их псевдонимов является частным случаем $\{\langle \text{описание} \rangle; \sim\}$ тогда и только тогда, когда он начинается с « $\sim \text{VAR} \sim$ », затем следует псевдоним переменной, потом – « $\sim :$ », потом тип, затем « $\sim ;$ », а заканчивает текст опять $\{\langle \text{описание} \rangle; \sim\}$. В терминах процедуры DECL и согласно индукционной гипотезе для правой части текста, последнее эквивалентно тому, что вызов процедуры DECL(правая_часть) завершается построением дерева грамматического разбора. Согласно уже доказанной тотальной корректности процедуры TYPE, вызов TYPE(тип_переменной) завершается построением дерева грамматического разбора типа тогда и только тогда, когда тип_переменной синтаксически корректный, что проверяется на шагах 2, 3 и 4 про-

цедуры DECL(текст). Следовательно, и в данном случае процедура DECL(текст) завершается построением дерева грамматического разбора тогда и только тогда, когда текст – это $\{\langle \text{описание} \rangle; \sim\}$.

Таким образом, доказательство тотальной корректности процедуры DECL завершено.

Остаётся доказать тотальную корректность модуля NeMoSyN. Предположим, что выполнены предусловия модуля. Согласно определению

$$\langle \text{программа} \rangle ::= \{\langle \text{описание} \rangle; \sim\} \langle \text{тело} \rangle,$$

всякое вхождение «;» в $\{\langle \text{описание} \rangle; \sim\}$ имеет глубину скобочной вложенности 0. Согласно определению

$$\langle \text{тело} \rangle ::= \langle \text{присваивание} \rangle \mid \langle \text{тест} \rangle \mid (\langle \text{тело} \rangle \sim \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle \cup \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle)^*,$$

всякое вхождение «;» в $\langle \text{тело} \rangle$ имеет глубину скобочной вложенности, по крайней мере, 1. Поэтому текст является $\langle \text{программой} \rangle$ тогда и только тогда, когда левая_часть (префикс вплоть до самого правого вхождения «;» с нулевым тэгом) является $\{\langle \text{описание} \rangle; \sim\}$, а правая_часть (суффикс после самого правого вхождения «;» с нулевым тэгом) является $\langle \text{телом} \rangle$. Следовательно, в силу доказанной тотальной корректности процедур DECL и BODY, имеем: текст является $\langle \text{программой} \rangle$ тогда и только тогда, когда DECL(левая_часть) и BODY(правая_часть) останавливаются, построив деревья грамматического разбора левой_части как $\{\langle \text{описание} \rangle; \sim\}$, а правой_часть – как $\langle \text{тело} \rangle$. Но именно это и проверяется в модуле NeMoSyN на шагах 3 и 4. ■

Сложность синтаксического анализа для NeMo

Утверждение 2 Алгоритмы NeMoLeX, TYPE, DECL, BODY и NeMoSyN имеют не более чем квадратичную оценку сложности (количества шагов) в зависимости от длины входного текста.

Доказательство проведём в порядке NeMoLeX, TYPE, BODY, DECL и NeMoSyN. Для NeMoLeX утверждение очевидно, так как алгоритм лексического анализа состоит в однократном прочтении по одному символу в порядке входного текста слева направо. Доказательство для TYPE и BODY аналогично друг другу получается индукцией по длине текста. Поэтому ниже мы разберём только доказательство для TYPE, а доказательство для BODY остаётся для самостоятельной работы. Доказательства для DECL и NeMoSyN получаются тривиальной комбинацией доказательств для TYPE и BODY аналогично друг другу.

Итак, разберём доказательство, что временная сложность (число шагов) процедуры TYPE оценивается сверху квадратичной функцией $f(n) = (n^2 + n + 1)$, где n – длина входного текста (т. е. количество символов в нём).

База индукции: пустой текст. Пустой текст не является типом в силу определения

$\langle \text{тип} \rangle ::= \text{INT} \mid \langle \text{тип} \rangle \sim \text{ARRAY} \sim \text{OF} \sim \langle \text{тип} \rangle.$

В то же время шаг 1 процедуры TYPE для пустого текста предписывает выдать сообщение об ошибке и на том завершить работу. Таким образом, доказательство базы индукции полностью исчерпано.

Индукционная гипотеза: предположим, что для всякого текста, длина которого не больше некоторого $m \geq 0$, если выполнены предусловия процедуры TYPE, то эта процедура завершает свою работу над текстом, выполнив не более $f(m) = m^2 + m + 1$ шагов.

Шаг индукции. Пусть текст состоит из $n = (m + 1)$ символов и удовлетворены предусловия процедуры TYPE. Так как текст не пуст, то шаг 1 процедуры неприменим. В силу определения имеем

$\langle \text{тип} \rangle ::= \text{INT} \mid \langle \text{тип} \rangle \sim \text{ARRAY} \sim \text{OF} \sim \langle \text{тип} \rangle.$

Если текст – это INT, то (согласно шагу 2 процедуры TYPE) данная процедура построит дерево грамматического разбора этого текста и завершит работу за один шаг, что очевидным образом не превосходит $f(n)$.

Если же текст – это не INT, то (согласно определению типа) он является типом тогда и только тогда, когда этот текст может быть представлен в виде

$(\text{тип_1 ARRAY OF тип_2}),$

где тип_1 и тип_2 – тоже типы. Процедура TYPE осуществляет за однократный просмотр текста проверку имеет ли он структуру

$(\text{текст_1 ARRAY OF текст_2})$

и выделяет одновременно текст_1 и текст_2, если данный текст имеет такую структуру. Если текст не имеет указанной структуры, то TYPE завершает работу, выполнив n шагов; в этом случае доказательство шага индукции завершено. Если же текст имеет указанную структуру, то процедура TYPE после выполнения этих n шагов просмотра рекурсивно вызывает TYPE(текст_1) и TYPE(текст_2), а потом комбинирует их результаты за один шаг (сгенерировав или сообщает об ошибке, или синтаксическое дерево). Пусть m_1 – длина слова_1, а m_2 – длина слова_2. Заметим, что $m_1 + m_2 < (n - 10)$, так как n – это длина всего слова (текст_1 ARRAY OF текст_2). Поэтому в соответствии с индукционной гипотезой общее число шагов процедуры TYPE на тексте оценивается сверху следующим образом:

$$n + f(m_1) + f(m_2) + 1 = n + ((m_1)^2 + m_1 + 1) + ((m_2)^2 + m_2 + 1) + 1 =$$

$= ((m_1)^2 + (m_2)^2 + n) + (m_1 + m_2 + 2) + 1 \leq n^2 + n + 1;$

следовательно, и в этом случае доказательство шага индукции завершено.

Таким образом, доказательство сложности процедуры TYPE закончено. ■

Следовательно, второе важное отличие описанного в лекции 9 лексического и синтаксического анализаторов от алгоритма Кока – Янгера – Касами состоит в эффективности: в нашем случае оценка сложности квадратичная вместо кубической для классического алгоритма.

Задача проверки контекстных зависимостей для языка HeMo

Для языка программирования, имеющего контекстно-свободный синтаксис и систему контекстных ограничений, построение дерева грамматического разбора текста, претендующего быть программой, ещё не позволяет утверждать, что перед нами действительно программа на этом языке.

Например, рассмотрим следующий текст на предмет, является ли он программой на нашем формализованном варианте языка HeMo:

VAR xxx : INT; VAR xxx : (INT ARRAY OF INT); (yy:= APP(3,2) ; 7=x ?).

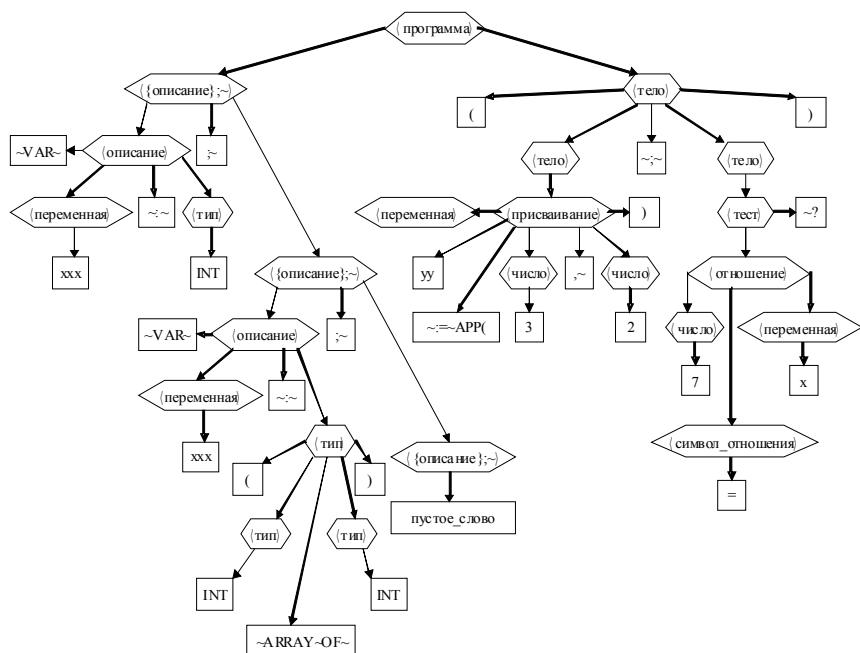
Модуль NeMoLeX преобразует его в следующий текст: «VAR @_{xxx} : INT;⁰ VAR @_{xxx} : (¹INT ARRAY OF¹ INT)^{1,0} (¹@_{yy}:= APP(@₃,@₂) ;¹ @₇=@_x ?)¹», где «@» с нижним индексом обозначает уникальный псевдоним, соответствующий переменной или числу, помещённому в индекс, а верхний числовой индекс обозначает значение тэга скобочной вложенности для «ARRAY OF», «<», «∪», «(», «)». Кроме того, этот модуль строит таблицу переменных TV

Переменная	Псевдоним	список типов
xxx	@ _{xxx}	INT, (INT ARRAY OF INT)
yy	@ _{yy}	
x	@ _x	

и таблицу констант TC

Число	псевдоним
3	@ ₃
2	@ ₂
7	@ ₇

В свою очередь модуль NeMoSyN построит следующее дерево грамматического разбора исходного текста в соответствии с контекстно-свободной грамматикой языка Немо:



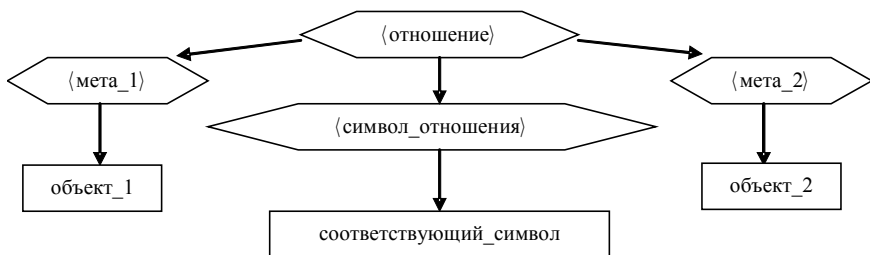
Таким образом, предъявленный текст соответствует понятию $\langle \text{программа} \rangle$ контекстно-свободного синтаксиса HeMo, но не является программой, поскольку нарушает сразу несколько запретов: переменная «xxx» описана дважды, а переменные «xx» и «у» вовсе не описаны.

Напомним список контекстных зависимостей для формализованного варианта языка HeMo:

- всякая переменная, встречающаяся в программе, должна иметь единственное описание;
- тип левой и правой части любого отношения в программе должен быть INT;
- типы левой и правой части любого присваивания в программе должны совпадать.

Первое условие проверяется тривиально: необходимо и достаточно проверить, что в каждой учётной записи в таблице переменных список типов не пуст и состоит из единственного типа, который и является типом этой переменной.

Второе условие также легко проверить. Для этого необходимо и достаточно обойти дерево грамматического разбора и для каждого поддерева следующего вида убедиться, что объект_1 и объект_2 является или $\langle \text{числом} \rangle$, или $\langle \text{переменной} \rangle$ типа INT:



На идейном уровне проверка третьего контекстного условия для языка НеМо тоже довольно проста: необходимо и достаточно для каждого присваивания (которые легко найти по дереву грамматического разбора) «вычислить» тип левой и правой части и сравнить их; тип левой части – это просто тип соответствующей переменной согласно таблице переменных; для правой части тип можно «вычислить» (или, правильнее сказать, «вывести») по следующим правилам вывода:

_____ , если переменная t
 t : type имеет тип type согласно
 таблице переменных

_____ , если n - число
 n : INT

t : (type₁ ARRAY OF type₂) , t₁ : type₁

APP(t, t₁) : type₂

t : (type₁ ARRAY OF type₂) , t₁ : type₁ , t₂ : type₂

UPD(t, t₁, t₂) : (type₁ ARRAY OF type₂)

Отметим, однако, что для реального языка программирования проверка контекстных ограничений может быть не только трудоёмкой, но и отнюдь не столь тривиальной задачей.

Лабораторная работа 2. Грамматический разбор и внутреннее представление НеМо-программ

Цель лабораторной работы:

- ознакомление на практике с основными понятиями лексического и синтаксического анализа контекстно-свободных языков;
- реализация учебного синтаксического анализатора для языка НеМо.

Что входит в лабораторную работу:

1. Формальное определение контекстно-свободного синтаксиса вашего диалекта языка НеМо в нотации Бэкуса – Наура или в форме синтаксических диаграмм.
2. Программа лексико-синтаксического анализа, которая строит дерево грамматического разбора для программ вашего диалекта НеМо и отвергает любой текст, не являющийся программой.

Требования к диалекту НеМо.

3. Поддержка, по крайней мере, двух типов данных INT и (INT ARRAY OF INT).
4. Поддержка многоместных конструкторов «;» и «⋈» для последовательного исполнения и недетерминированного выбора.
5. Поддержка стандартных средств доступа к элементам массивов и обновления массивов или APP и UPD.
6. Поддержка выражений с одним и/или двумя операндами в правых частях операторов присваивания и в условиях тестов.
7. Поддержка отношений «=», «<», «>» «<=» и «>=» без булевских операций в условиях тестов.
8. Поддержка не менее чем 26 переменных, соответствующих строчным буквам латинского алфавита.
9. Выделение пробелами и/или переводами строки ключевых слов («VAR», «INT», «ARRAY OF» и т. д.) и конструкторов («;» и «U»).
10. Выделение пробелами и/или переводами строки отдельных описаний и отделение описаний от тела программы посредством перевода строки.

Специальных требований к представлению алгоритма лексико-синтаксического анализа нет. Описание алгоритма может быть чисто словесным (примерно полстраницы), в форме хорошо структурированного текста (около полутора страниц), в виде псевдокода и/или блок-схем (объемом до трёх страниц) или выполнено с использованием всех этих средств (до трёх страниц). Наличие описания алгоритма обязательно, а разумная структуризация и человеко-дружественная формализация приветствуются.

Требования к программе лексико-синтаксического анализа.

1. Предусловие: входные данные – ASCII файл с расширением NEM.
2. Постусловия:
 - 2.1) Если входной файл не является программой на вашем диалекте HeMo, то формируется LOG-файл с соответствующим сообщением (желательно с указанием, что именно не соответствует синтаксису).
 - 2.2) Если входной файл является программой на принятом диалекте HeMo, то формируется «таблица» переменных программы и «дерево» грамматического разбора тела программы.
 - 2.2.1) «Таблица» переменных представляет собой базу данных, которая для каждой описанной переменной устанавливает её тип, «внешнее» имя (определённое в программе), и «внутреннее» имя (используемое вместо внешнего), причём как «внешнее», так и «внутреннее» имя являются ключами «таблицы».

2.2.2) «Дерево» грамматического разбора тела программы представляет собой базу данных, в которой вместо «внешних» имён переменных используются их «внутренние» имена и которая по каждой подстроке, самой по себе являющейся телом, предоставляет следующую информацию:

2.2.2.1) если подстрока это оператор присваивания, то информация включает признак (флаг) присваивания, имя переменной в левой части, операцию и имена операндов в правой части;

2.2.2.2) если подстрока это тест, то информация включает признак (флаг) теста, отношение, операцию и имена операндов в левой части, операцию и имена операндов в правой части;

2.2.2.3) если подстрока это последовательная композиция («;»)
нескольких подтел, то информация включает признак (флаг) последовательной композиции и «ссылки» на все подтела, упорядоченные в соответствии с их порядком в подстроке слева направо;

2.2.2.4) если подстрока это недетерминированный выбор («U»)
из нескольких подтел, то информация включает признак (флаг) недетерминированного выбора и «ссылки» на все подтела, упорядоченные в соответствии с их порядком в подстроке слева направо;

2.2.2.5) если подстрока это недетерминированный цикл («*»),
то информация включает признак (флаг) недетерминированного цикла и «ссылку» на тело этого цикла.

Комментарии и рекомендации.

1. Формальное определение синтаксиса – это фактически мини-контракт между «заказчиком» и «разработчиком» транслятора с HeMo. «Заказчик» сформулировал свои требования к синтаксису на неформальном уровне в первой лекции курса, а вариант формализации синтаксиса – в шестой лекции. Формализованный «разработчиком» вариант синтаксиса должен быть выполнен в дружественной форме для «заказчика», на пути формализации его понятий. Например, если уж «заказчик» говорит о теле программы, то в формальном синтаксисе надо определять метасимвол ⟨тело⟩, а не какой-либо ⟨оператор⟩, ⟨блок⟩ или ещё что-либо «аналогичное». Или если символ «;» в теле программы используется для последовательной композиции, то не надо использовать этот же символ в теле для обозначения конца операторов присваивания.

2. «Дерево» грамматического разбора тела программы возможно построить на основе алгоритма Янгера – Кока – Касами. Но этот путь доказывает только теоретическую возможность построения и на практике неэффективен (по крайней мере, для НеМо), поскольку алгоритм строит все возможные деревья грамматического разбора снизу вверх (начиная с односимвольных подстрок и заканчивая всей строкой), а нам нужно одно «дерево» и в порядке сверху вниз (от строки – к её подстрокам и т. д.). Поэтому для НеМо с его простой синтаксической структурой необходимо разработать и представить свой специализированный эффективный алгоритм построения «дерева» грамматического разбора тела программы.
3. Соответствие реализации разработанному алгоритму остаётся на «разработчике». Выбор языка реализации тоже полностью зависит от «разработчика». Допустим выбор любого языка, в том числе который лучше знаком «разработчику» или, наоборот, с которым «разработчик» хочет лучше познакомиться. Однако можно посоветовать выбрать один из функциональных или логических языков программирования (например, классические Рефал, LISP, ML и Prolog), к сожалению, плохо знакомых большинству студентов. Тем не менее, есть веские причины для выбора одного из этих языков: рекомендуемые языки имеют мощные встроенные механизмы сопоставления с образцом (Рефал, LISP и ML) или унификации (Prolog), что сильно упрощает программирование выбора применимого правила грамматики при лексическом и синтаксическом анализе и сокращает общий объем программы в разы по сравнению с C(++) и Java.

Часть III. СЕМАНТИКА И ТРАНСЛЯЦИЯ

Лекция 11. Семантика типов данных (на примере языка HeMo)

Зачем нужна формальная семантика языков

Неформальное описание языка HeMo заняло у нас очень немного места – менее половины первой лекции. Выше мы уже убедились, что неформально описанный синтаксис языка HeMo может быть формализован несколькими способами, с использованием разных нотаций, но главное – в виде разных синтаксических диалектов языка. Эти разные диалекты могут быть просто несовместимы. Но для того, чтобы однозначно определить конкретный синтаксический диалект HeMo на человеческом языке (без использования формализмов грамматик, нотации Бэкуса – Наура или синтаксических диаграмм) нам пришлось бы посвятить отдельную лекцию пространным и неоднозначным описаниям. Более того, исходя из таких неформальных определений было бы крайне сложно сконструировать алгоритм проверки синтаксической правильности программ на HeMo, а также доказать корректность этого алгоритма.

Аналогично обстоит дело и с формализацией семантики (смысла) программ на любом языке программирования вообще и на языке HeMo в частности. Без формализации семантики крайне затратно (по времени и размерам) давать ясное и однозначное определение смысла программ, определять язык спецификаций свойств программ, доказывать вручную соответствие программ спецификациям их свойств. Без формализации семантики задача верифицирующего транслятора становится вообще нереальной до создания мыслящей машины, способной воспринимать и формализовывать неформализованную информацию. Другими словами, без формализации семантики программ попытки решения проблемы верифицирующего компилятора надо отложить до момента, когда ЭВМ успешно «сдадут» тест А. Тьюринга.

Мы не будем откладывать решение задачи верифицирующего транслятора для языка HeMo столь надолго, а потому формализуем семантику нашего модельного языка. Но прежде обсудим два примера, которые иллюстрируют значение формальной семантики.

Пример первый. Клавиатура компьютера имеет конечный набор символов, отображаемых на экране. Можно считать, что число таких символов не

больше 200. Цепочками, состоящими из 66 символов, можно взаимно-однозначно закодировать все целые числа в интервале $[1..66^{200}]$. Тогда какое число кодирует следующая цепочка⁴⁸: «Это предложение кодирует число на 1 большее чем 66 в степени 200.». Сама эта цепочка состоит из 66 отображаемых символов, поэтому она кодирует некоторое число из $[1..66^{200}]$. Но в то же время по смыслу эта цепочка кодирует число $(1 + 66^{200})$. Парадокс? Разумеется, нет. Просто речь идёт о двух разных семантиках цепочек. В одном случае это чисто «механическая» семантика кодировок чисел, в другом – «человеческая» семантика естественного языка.

Пример второй. Предположим, дана задача в следующей формулировке: найти z в $z^2 + 1 = 0$. У математически «грамотного» человека⁴⁹ сразу возникает вопрос, в каких числах надо решить это уравнение? Если в вещественных числах, то уравнение не имеет решений; если же речь идёт о комплексных числах, то уравнение имеет единственное решение – комплексную единицу i .

У математически «образованного» человека возникнет вопрос: может быть, речь идёт о решении уравнения в кольце вычетов по модулю некоторого натурального числа n , например $n=3$ или $n=5$? В первом случае уравнение не имеет корней, а во втором случае корень этого уравнения $x=3$:

1. $0^2 + 1 = 1 \neq 0 \pmod 3$, $1^2 + 1 = 2 \neq 0 \pmod 3$, $2^2 + 1 = 5 \neq 0 \pmod 3$;
2. $3^2 + 1 = 10 = 2 \times 5 + 0 = 0 \pmod 5$.

У математически «продвинутого» человека может возникнуть вопрос: а может быть речь идёт вообще о каком-нибудь абстрактном кольце? Действительно, в уравнении $z^2 + 1 = 0$ использованы сложение, умножение (или возведение в степень), нулевой и единичный элементы, которые в любом кольце принято обозначать «+», «×» (для возведения в квадрат), «0» и «1» соответственно. Но в такой общей постановке остаётся только развести руками, так как ничего конкретного про решение уравнения $z^2 + 1 = 0$ сказать невозможно.

Однако откуда следует, что задача вообще состоит в решении уравнения? Её формулировка «найти z в $z^2 + 1 = 0$ » вообще не использует слов «уравнение» и «решить»! В этом случае неявно использовалась «стандартная математическая семантика» фразы «найти z в $z^2 + 1 = 0$ », означающая

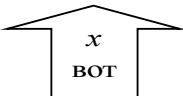
⁴⁸ Кавычки использованы для выделения цепочки и не входят в её состав.

⁴⁹ В курсе не предполагается глубоких знаний теории комплексных чисел или теории вычетов натуральных чисел. Однако подразумевается, что базовые определения известны:

- комплексная единица i – это единственный корень уравнения $z^2+1=0$;
- кольцо вычетов по модулю натурального числа n – это множество $[0..(n-1)]$ остатков от деления на n с операциями сложения, вычитания, умножения и (когда возможно) деления.

«решить уравнение $z^2 + 1 = 0$ относительно z ». Но даже явное осознание неявного использования «стандартной математической семантики» не спасает положения: как видно из анализа стандартной математической семантики задачи на решение уравнения, задача всё ещё семантически недоопределена.

Заметим также, что кроме «стандартной математической семантики» для задачи «найти z в $z^2 + 1 = 0$ » могут существовать и «нестандартные семантики». Например, нестандартным возможным решением задачи «найти z в $z^2 + 1 = 0$ » может быть следующее:

$$x^2 + 1 = 0$$


В этой семантике $z^2 + 1 = 0$ интерпретируется как строка символов, а z интерпретируется как символ.

«Наивная» семантика типов данных HeMo

Язык HeMo имеет единственный предопределённый тип данных INT и единственный родовой тип (ТИП1 ARRAY OF ТИП2). С предопределённым типом INT всё кажется простым и ясным: его семантика – это множество математических целых чисел с обычными математическими операциями «+», «-», «×», «/» и отношениями «=», «≠», «<», «>», «≤», «≥» на них. Сложнее обстоит дело с родовым типом как таковым и конкретными примерами типа массив, например, (INT ARRAY OF INT), (INT ARRAY OF (INT ARRAY OF INT)), ((INT ARRAY OF INT) ARRAY OF INT) и т. д., которые получаются из предопределённого типа при помощи родового типа.

«Наивный» взгляд на массивы выглядит примерно так. Значение типа (INT ARRAY OF INT) – это таблица из бесконечного числа занумерованных элементов, каждый из которых может быть или целым числом, или неопределённым значением. Следовательно, значение типа (INT ARRAY OF (INT ARRAY OF INT)) – это таблица из бесконечного числа занумерованных элементов, каждый из которых имеет тип (INT ARRAY OF INT), т. е. может быть или неопределённым значением, или таблицей из бесконечного числа занумерованных элементов, каждый из которых в свою очередь может быть целым числом или неопределённым значением. Иными словами,

значение (INT ARRAY OF (INT ARRAY OF INT)) – это двухмерная таблица, бесконечная по обоим измерениям, каждый элемент которой может быть или целым числом, или неопределённым значением. А самое большее, на что способен «наивный» подход, – это интерпретировать значение типа

INT ARRAY OF (INT ARRAY OF ... (INT ARRAY OF INT) ...)

как многомерную «таблицу» бесконечную по всем измерениям, каждый элемент которой в свою очередь может быть или целым числом, или неопределённым значением. Но интерпретировать или придать смысл значению типа ((INT ARRAY OF INT) ARRAY OF INT) «наивный» подход не в состоянии.

Правда, «наивный» подход может объяснить для тех массивов, которые с его точки зрения вполне «легальные» таблицы, что такое элемент массива и что такое присваивание элементу массива: это соответственно взятие элемента таблицы и изменение значения элемента таблицы. Кроме того, такой подход может допустить сравнение массивов как таблиц на совпадение (если только возможно перебрать бесконечное множество индексов и сравнивать неопределённые значения между собой).

«Математическая» семантика типов данных HeMo

Математический подход на основе неформальной теории множеств к семантике типов данных способен придать вполне определённый смысл не только значениям типа ((INT ARRAY OF INT) ARRAY OF INT), но и значениям любого конкретного типа языка HeMo и родовому типу (ТИП1 ARRAY OF ТИП2) следующим образом.

Определение 1. Семантика всякого конкретного типа с точки зрения неформальной теории множеств – это множество значений этого типа вместе с совокупностью частичных или тотальных операций, в которых эти значения могут быть аргументами или результатами, и совокупность отношений, в которых эти значения могут участвовать. Семантика родового типа с точки зрения неформальной теории множеств – это (частичная) функция, которая по конкретным типам строит новый тип, т. е. по множествам значений с допустимыми операциями и отношениями строит «новое» множество значений со своими операциями и отношениями, в которых могут использоваться эти «новые» значения.

Определение. Математическая семантика типа INT языка HeMo определяется следующим образом. Множество значений типа INT – это множество целых чисел \mathbf{Z} . Следующие нольместные функции (константы) ..., -2, -1, 0, 1, 2, ... имеют значения в этом множестве. А двуместные тотальные

функции (операции) «сложение» = $\lambda x, y \in \mathbf{Z}. (x + y)$, «вычитание» = $\lambda x, y \in \mathbf{Z}. (x - y)$ и «умножение» = $\lambda x, y \in \mathbf{Z}. (x \times y)$, а также частичная функция «деление» = $\lambda x, y \in \mathbf{Z}. (x / y)$ имеют все аргументы и результаты в этом же множестве. Кроме того, значения данного типа могут быть аргументами операции взятия элемента массива с индексами типа INT, изменения значения элемента массива с индексами типа INT, нового значения элемента массива со значениями типа INT (подробнее эти операции будут обсуждаться ниже вместе с массивами). Значения этого типа могут быть аргументами следующих бинарных отношений: «равно» = $\{ (x, x) : x \in \mathbf{Z} \}$, «меньше» = $\{ (x, y) : x, y \in \mathbf{Z} \text{ и } x < y \}$, «больше» = $\{ (x, y) : x, y \in \mathbf{Z} \text{ и } x > y \}$, «небольше» = $\{ (x, y) : x, y \in \mathbf{Z} \text{ и } x \leq y \}$, «неменьше» = $\{ (x, y) : x, y \in \mathbf{Z} \text{ и } x \geq y \}$.

Определение. Математическая семантика родового типа (ТИП1 ARRAY OF ТИП2) языка HeMo определяется следующим образом. Множество значений этого типа – это множество $(D2 \cup \{\text{неопределено}\})^{D1}$ всех частичных функций $f: D1 \rightarrow D2$, где $D1$ – множество значений для ТИП1, а $D2$ – множество значений для ТИП2. Математическая семантика взятия элемента массива – это операция APP: $(D2 \cup \{\text{неопределено}\})^{D1} \times D1 \rightarrow D2$ применения функции к аргументу, т. е. вычисления значения функции в точке, которая формально определяется следующим образом: $\lambda f \in (D2 \cup \{\text{неопределено}\})^{D1}. \lambda x \in D1. (f(x))$. А математическая семантика определения нового значения для элемента массива – это операция UPD: $(D2 \cup \{\text{неопределено}\})^{D1} \times D1 \times D2 \rightarrow (D2 \cup \{\text{неопределено}\})^{D1}$, возвращающая новую функцию, совпадающую с исходной всюду, кроме точки, для которой определяется новое значение, т. е. UPD – это операция

$$\lambda f \in (D2 \cup \{\text{неопределено}\})^{D1}. \lambda x \in D1. \lambda y \in D2.$$

$$(\lambda z \in D1. \text{if } z=x \text{ then } y \text{ else } f(z)).$$

Значения типа (ТИП1 ARRAY OF ТИП2) не участвуют ни в каких отношениях ни между собой, ни со значениями других типов.

Такая формализация значений типа массив естественно обобщает «наивную» семантику массивов как таблиц, поскольку таблицы – это не более чем один из форматов задания функции, у которых аргументы – картежи целых чисел. Она также обобщает «наивный» табличный вариант элемента массива: для табличной функции вычисление её значения по аргументу – это просто соответствующий элемент таблицы. Математическая формализация изменения значения элемента массива также обобщает «наивный» табличный вариант: для табличной функции изменение значения её элемента – это новая таблица, совпадающая с исходной всюду, кроме одного обновляемого элемента.

Конечно, в математической семантике для типа (ТИП1 ARRAY OF ТИП2) можно определить теоретико-множественные отношения, например, «равно» = $\{ (f, f) : f: D1 \rightarrow D2 \}$, «шире» = $\{ (f, g) : f, g: D1 \rightarrow D2, f \subseteq g \}$ и «уже» = $\{ (f, g) : f, g: D1 \rightarrow D2, g \subseteq f \}$, или бинарные отношения, которые поэлементно наследуются от ТИП2, например, «поэлементное меньше» = $\{ (f, g) : f, g: D1 \rightarrow D2 \text{ и } f(x) < g(x) \text{ для любого } x \in D1 \}$, если для ТИП2 определено отношение «<», или «поэлементное больше» = $\{ (f, g) : f, g: D1 \rightarrow D2 \text{ и } f(x) > g(x) \text{ для любого } x \in D1 \}$, если для ТИП2 определено отношение «>», и т. д. Также теоретически возможно для типа (ТИП1 ARRAY OF ТИП2) определить и другие операции, которые уже трудно интерпретировать с «наивной» точки зрения (например, можно определить оператор «рекурсии по массиву» как самого обычного рекурсивного определения функции, но это уже выходит за рамки курса).

Математическая (теоретико-множественная) семантика позволяет интерпретировать те массивы, на которых «споткнулась» наивная семантика типов данных. Так, например, значения типа ((INT ARRAY OF INT) ARRAY OF INT) – это функции, сопоставляющие (частичным) целочисленным функциям (INT ARRAY OF INT) целые числа (INT), т. е. функционалы (второго порядка). Пример такого функционала – $\lambda f \in (\mathbf{Z} \cup \{\text{неопределено}\})^{\mathbf{Z}}. (f(0))$, сопоставляющий каждой частичной целочисленной функции $f: \mathbf{Z} \rightarrow \mathbf{Z}$ её значение $f(0)$ в точке 0; иными словами, функционал $\lambda f \in (\mathbf{Z} \cup \{\text{неопределено}\})^{\mathbf{Z}}. (f(0))$ – пример массива типа ((INT ARRAY OF INT) ARRAY OF INT). А значения другого типа ((INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT)) – это функции, сопоставляющие (частичным) целочисленным функциям (INT ARRAY OF INT) (частичные) целочисленные функции (INT ARRAY OF INT), т. е. операторы (второго порядка). Пример такого оператора – λf полином на \mathbf{Z} . (производная f'), который сопоставляет каждому полиному $f: \mathbf{Z} \rightarrow \mathbf{Z}$ его производную f' ; иными словами, функционал λf полином на \mathbf{Z} . (производная f') – пример массива типа (INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT).

Зачем нужны другие семантики?

Математическая семантика типов данных на основе неформальной теории множеств обладает рядом достоинств:

- преемственностью с неформальной семантикой,
- интерпретацией очень сложных типов,
- непротиворечивостью на уровне неформальной теории множеств;
- открытостью для дальнейших обобщений и расширений.

Однако, у математической семантики есть и существенные недостатки. Один из них – зависимость от человеческих представлений о том, что такое множество. «Множество есть совокупность, мыслимая как единое», – «определение» понятия множества, положенное Георгом Кантором в основание заложенной им неформальной теории множеств, к сожалению, «непереводимо» для компьютера. Но именно компьютер должен автоматически (без вмешательства человека) исполнять и верифицировать программы, которые могут оперировать со сколь угодно сложными типами. Другой важный недостаток – слишком большой «разброс» возможных значений допустимых типов, которые просто нельзя «хранить» обычным образом в компьютере как «таблицу»: невозможно, например, хранить в виде таблицы массив $(\lambda f \text{ полином на } \mathbb{Z}. (\text{производная } f'))$ типа $((\text{INT ARRAY OF INT}) \text{ ARRAY OF } (\text{INT ARRAY OF INT}))$, в котором каждой функции, заданной полиномом с целыми коэффициентами, сопоставляется её производная функция⁵⁰.

Поэтому надо предложить два «согласованных» вида семантики для типов данных, один из которых ориентирован на верификацию программ (т. е. позволяет верифицирующему транслятору «рассуждать» о свойствах значений разных типов данных), а другой – на исполнение программ (т. е. позволяет верифицирующему транслятору преобразовывать значения разных типов данных).

Аксиоматическая семантика типов данных HeMo

Аксиоматическая семантика призвана дать верифицирующему транслятору возможность «рассуждать» о свойствах значений типов данных на чисто синтаксическом уровне, не апеллируя к человеческому представлению о том, что такое множество, функция и т.д.

Определение 2. Аксиоматический подход к семантике типов данных состоит в том, что для каждого из базисных (предопределённых) типов данных формулируется своя аксиоматическая теория первого порядка, для каждого из родовых типов формулируются свои (схемы) правил, позволяющих построить аксиоматическую теорию любого конкретного типа, построенного из базисных типов посредством родовых типов.

Ниже разберём аксиоматический подход на примере типов языка HeMo.

Единственный предопределённый тип данных HeMo – INT. Значения его – это ноль, натуральные числа и им противоположные отрицательные числа. Для натуральных чисел существует общепризнанная аксиоматизация

⁵⁰ Очевидно, что в данном случае надо хранить не таблицу пар (функция, производная), а метод, который по произвольной полиномиальной функции строит её производную.

Дж. Пеано, состоящая из общелогических аксиом и правил вывода (см., например, систему ВХ из лекции 4), и специальных арифметических аксиом 1 – 6 и правила вывода IND, которые перечислены ниже в стиле, который восходит к Д. Гилберту:

- 1) $0 \neq (x + 1)$;
 - 2) $(x + 1) = (y + 1) \rightarrow x = y$;
 - 3) $x + 0 = x$;
 - 4) $x + (y + 1) = (x + y) + 1$;
 - 5) $x \times 0 = 0$;
 - 6) $x \times (y + 1) = (x \times y) + x$;
- IND:
$$\frac{\varphi(0) \quad , \quad \varphi(x) \rightarrow \varphi(x + 1)}{\forall x. \varphi(x)} \quad .$$

Для совместимости с нотацией, принятой нами в лекции 4 для аксиоматической системы ВХ, все аксиомы 1 – 6 переписываются в виде одной аксиомы

$$\frac{}{\varphi, \Sigma} \quad ,$$

где φ – это одна из аксиом 1 – 6, а правило вывода IND принимает следующий вид:

$$\frac{\varphi(0), \Sigma \quad (\neg\varphi(x) \vee \varphi(x+1)), \Sigma}{(\forall x. \varphi(x)), \Sigma} \quad .$$

Преобразуем аксиоматику Дж. Пеано в аксиоматизацию типа INT следующим образом. Кроме используемых в аксиоматике Дж. Пеано символов «=» и « \neq » для двуместных отношений «равно» и «неравно», символов «0» и «1» для констант «ноль» и «один», символов «+» и « \times » для двуместных операций «сложить» и «умножить», используем ещё специальный символ « $\in \text{INT}$ » для одноместного отношения «быть значением типа INT», символы «<», «>», « \leq » и « \geq » для двуместных отношений «меньше», «больше», «небольше» и «неменьше» на значениях типа INT, символ «-» для одноместной операции «противоположное число», символы «-» и «/» для двуместных операций «вычитание» и «деление нацело» на значениях типа INT. Тогда аксиоматическая система для рассуждений о свойствах значений типа INT состоит из:

- общелогических аксиом и правил вывода;

- аксиомы $0 \in \text{INT} \ \& \ 1 \in \text{INT}$, определяющей, что «0» и «1» являются значениями типа INT;
- аксиом $\forall x, y \in \text{INT}. (x \leq y \leftrightarrow \exists z \in \text{INT}. (0 \leq z \ \& \ y = (x + z)))$, $\forall x, y \in \text{INT}. ((x - y) = (x + (-y)))$ и т. д., определяющих « \leq », « $>$ », « \leq », « \geq », « $-$ » и « $/$ » через « $+$ », « $-$ », « \times » и « \div »;
- аксиом $\forall x. (x \in \text{INT} \leftrightarrow -x \in \text{INT})$ и $\forall x \in \text{INT}. (x \geq 0 \vee -x \geq 0)$, определяющих, что каждое значение типа INT или ему противоположное значение типа INT является натуральным числом;
- аксиоматической системы Дж. Пеано для неотрицательных значений типа INT.

Единственный родовой тип данных HeMo – массив ARRAY OF. Как уже было сказано, со всяким родовым типом в аксиоматической семантике типов данных связаны правила, позволяющие добавлять новые аксиомы и правила вывода для значений всякого нового типа, конструируемого при помощи этого родового типа из уже построенных и аксиоматизированных типов данных. Для того, чтобы «плавно» подойти к такой аксиоматической семантике родового типа массив ARRAY OF, рассмотрим сначала аксиоматику массивов некоторого конкретного типа (ТИП1 ARRAY OF ТИП2).

Определение 3. Предположим, что аксиоматическая семантика для type1 и type2 уже построена, причём одноместные отношения « $\in \text{type1}$ » и « $\in \text{type2}$ » – «значение имеет тип type1» и «значение имеет тип type2» аксиоматизированы. Тогда аксиоматика для значений типа (type1 ARRAY OF type2) будет использовать новый символ « $\text{new}_{[\text{type1} \rightarrow \text{type2}]}$ » для нульместной операции «создание нового значения типа (type1 ARRAY OF type2)», новый символ « $\in [\text{type1} \rightarrow \text{type2}]$ » для одноместного отношения «значение имеет тип (type1 ARRAY OF type2)», а также специальный символ « $\text{APP}_{[\text{type1} \rightarrow \text{type2}]}$ » для двуместной операции «вычисления элемента массива типа (type1 ARRAY OF type2)» и специального символа « $\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}$ » для трёхместной операции «обновления элемента массива типа (type1 ARRAY OF type2)»⁵¹. Тогда аксиоматическая система для рассуждений о свойствах значений типа (type1 ARRAY OF type2) включает:

- общелогические аксиомы и правила вывода;
- аксиому $\text{new}_{[\text{type1} \rightarrow \text{type2}]} \in [\text{type1} \rightarrow \text{type2}]$, определяющую, что « $\text{new}_{[\text{type1} \rightarrow \text{type2}]}$ » является значением типа (type1 ARRAY OF type2);
- аксиому $\forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. (\text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x) \in \text{type2})$, определяющую, что $\text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x)$ является значением типа type2, и

⁵¹ В тех случаях, когда тип (type1 ARRAY OF type2) очевиден из контекста, словосочетание «типа (type1 ARRAY OF type2)» может быть опущено в названиях операций и отношений «создание нового значения типа (type1 ARRAY OF type2)» и т. д.

аксиому $\forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. \forall y \in \text{type2}. (\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y) \in [\text{type1} \rightarrow \text{type2}])$, определяющую, что « $\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y)$ » является значением типа ($\text{type1 ARRAY OF type2}$);

- аксиому $\forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. \forall y \in \text{type2}. (y = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y), x))$, аксиому $\forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x', x'' \in \text{type1}. \forall y \in \text{type2}. (x' \neq x'' \rightarrow \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x'') = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x', y), x''))$ и аксиому $\forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. (f = \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x)))$, совместно определяющих, как изменяется массив после обновления его элемента.

%%%

Определение 1

Аксиоматическая семантика родового типа массив ARRAY OF – это следующие правила генерации аксиом для каждого конкретного типа массив:

- $\lambda \text{ type1, type2. new}_{[\text{type1} \rightarrow \text{type2}]} \in [\text{type1} \rightarrow \text{type2}]$,
- $\lambda \text{ type1, type2. } \forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. (\text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x) \in \text{type2})$,
- $\lambda \text{ type1, type2. } \forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. \forall y \in \text{type2}. (\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y) \in [\text{type1} \rightarrow \text{type2}])$,
- $\lambda \text{ type1, type2. } \forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. \forall y \in \text{type2}. (y = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y), x))$,
- $\lambda \text{ type1, type2. } \forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x', x'' \in \text{type1}. \forall y \in \text{type2}. (x' \neq x'' \rightarrow \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x'') = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x', y), x''))$,
- $\lambda \text{ type1, type2. } \forall f \in [\text{type1} \rightarrow \text{type2}]. \forall x \in \text{type1}. (f = \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x)))$.

Полнота и неполнота аксиоматической семантики типов данных НеМо

Приведённая выше аксиоматическая семантика не определяет явно первый или высший порядок теории типов данных: все приведённые аксиомы и правила вывода могут трактоваться как первого порядка (когда все массивы интерпретируются некими однородными значениями, различающимися по типам только благодаря отношениям « $\in \text{type}$ »), так и высшего порядка (когда массивы интерпретируются соответствующими функциями, функциями от функций, и так далее). Проанализируем достоинства и недостатки обоих вариантов.

Если принять высший порядок аксиоматической семантики типов данных, то мы получаем непротиворечивую, но неполную теорию. «Непротиворечивая теория» в данном случае означает, что всё, что будет доказуемо в этой теории, будет автоматически верным для стандартной интерпретации,

в которой INT – это целые числа с соответствующими операциями и отношениями, (INT ARRAY OF INT) – частичные целочисленные функции над целыми числами, и так далее. А «неполнота теории» в данном случае означает, что (в силу теоремы неполноты, о которой будет идти речь подробнее ниже) невозможно аксиоматизировать непротиворечивым образом все свойства целых чисел, и, следовательно, все свойства всех функций высших порядков, которые строятся исходя из множества целых чисел.

Если же принять первый порядок аксиоматической теории типов данных, то (в силу отмеченной выше непротиворечивости) верифицирующий компилятор имеет возможность устанавливать посредством чисто синтаксического вывода свойства, которые истинны во всех моделях первого и высшего порядка, в которых выполнены все аксиомы. В тоже время имеют место две следующие теоремы, принадлежащие К. Гёделю:

- (1) Теорема о полноте⁵² теорий первого порядка: если какое-либо свойство первого порядка имеет место во всех моделях первого порядка, в которых выполнены все аксиомы аксиоматической теории первого порядка, то это свойство доказуемо в этой аксиоматической теории.
- (2) Теорема о неполноте формальной арифметики: для любой непротиворечивой аксиоматической системы, которая расширяет аксиоматику Дж. Пеано, существует такое свойство первого порядка, что ни это свойство, ни его отрицание не доказуемы в данной аксиоматической системе.

Теорема К. Гёделя о неполноте означает, в частности, что невозможно аксиоматизировать непротиворечивым образом все истинные утверждения для целых чисел с операциями «+», «-», « \times » и «/», и, поэтому наша аксиоматизация типа INT и всех построенных из INT массивов неполна и в принципе не может быть полной. Следовательно, любой верифицирующий компилятор для НеМО всегда будет иметь риск столкнуться с утверждением, которое он не может доказать, хотя это утверждение на самом деле истинно. С точки зрения теоремы К. Гёделя о полноте это означает, что это истинное утверждение опровергается в некоторой «нестандартной» модели первого порядка, где «целые числа» – это не только математические целые числа, а «массивы» – это не только функции. Существование очень даже необычных нестандартных моделей следует в частности из теоремы⁵³ Л. Лёвенгейма и Т. Сколема, которая утверждает, что всякая непротиворечивая теория первого порядка, использующая не более чем счётный алфавит символов, имеет счётную модель. Для аксиоматической семантики первого

⁵² См. утверждение 6 в лекции 4.

⁵³ См. утверждение 7 в лекции 4.

порядка типов данных языка HeMo это означает, что имеется модель, в которой, например, массивов типа (INT ARRAY OF INT) только счётное множество, в то время как в математической семантике все эти массивы интерпретируются частичными функциями $f: \mathbf{Z} \rightarrow (\mathbf{Z} \cup \{\text{неопределено}\})$, а таких функций существует континуум...

Понятие алгебраическая семантика типов данных

Частным случаем аксиоматической семантики первого порядка для типов данных является аксиоматизация типов данных посредством тождеств или условных тождеств. Прежде чем определять эти понятия в общем виде, остановимся на простом математическом примере.

В алгебре группа – это любое множество с тремя операциями: нульместной «единица», одноместной «обращение» (суффиксной) и двуместной «умножение» (инфиксной). Для них используются символы «e», « $^-$ » и « \cdot » соответственно). Для любых элементов группы выполняются следующие тождества, известные как аксиомы теории групп:

1. $e \cdot x = x \cdot e = x$,
2. $(x^-) \cdot x = x \cdot (x^-) = e$,
3. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Если кроме константы «e» выделены ещё несколько констант «a», «b», «c» и так далее, то любая группа, в которой эти константы интерпретированы элементами этой группы, называется группой с выделенными элементами «a», «b», «c», ... Среди всех таких групп особое место принадлежит так называемой группе свободнопорождённой константами «a», «b», «c», ... Эта свободнопорождённая группа определяется следующим образом:

- элементы этой группы – единица «e» и все слова, построенные из «a», «a $^-$ », «b», «b $^-$ », «c», «c $^-$ » и т.д., в которых не встречается «контрарных пар», т.е. подслов вида «aa $^-$ » и «a $^-$ a», «bb $^-$ » и «b $^-$ b», «cc $^-$ » и «c $^-$ c», и т.д.;
- единица в этой группе – это единица «e»;
- операция обращения в этой группе определяется рекурсивными равенствами $e^- = e$, $(x \cdot w)^- = (w)^- x^-$, $(x^-)^- = x$, «пусто» $^-$ = «пусто», где x – это «a», «a $^-$ », «b», «b $^-$ », «c», «c $^-$ » и так далее, а w – любое слово, построенное из «a», «a $^-$ », «b», «b $^-$ », «c», «c $^-$ » и так далее;
- операция произведения в этой группе определяется как $e \cdot x = x \cdot e = x$ для любого элемента x этой группы, или следующим алгоритмом:

{ $x = w'$ и $y = w''$, где w' и w'' - слова, построенные из «a», «a $^-$ », «b», «b $^-$ », «c», «c $^-$ » и так далее без контрарных пар} // предусловие
 $w := w'w''$;

WHILE w имеет контрарную пару DO $w := \text{стереть } w \text{ в } w$ контрарную пару ;

IF $w = \langle \text{пусто} \rangle$ THEN $(x \cdot y) := e$ ELSE $(x \cdot y) := w$

$\{(x \cdot y) = e \text{ или } (x \cdot y) = w, \text{ где } w - \text{слово, построенное } \langle a \rangle, \langle a^- \rangle, \langle b \rangle, \langle b^- \rangle, \langle c \rangle, \langle c^- \rangle \text{ и так далее без контрарных пар}\} // \text{постусловие.}$

На языке теории программирования

- совокупность всех групп с выделенными элементами $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ называется «абстрактным типом данных с порождающими элементами $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ и тождествами $e \cdot x = x \cdot e = x, (x^-) \cdot x = x \cdot (x^-) = e$ и $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ »;
- каждая конкретная группа с выделенными элементами $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ называется «реализацией абстрактного типа данных с порождающими элементами $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ и тождествами $e \cdot x = x \cdot e = x, (x^-) \cdot x = x \cdot (x^-) = e$ и $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ »;
- группа, свободнопорождённая $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ называется «инициальной алгеброй с порождающими элементами $\langle a \rangle, \langle b \rangle, \langle c \rangle, \dots$ и тождествами $e \cdot x = x \cdot e = x, (x^-) \cdot x = x \cdot (x^-) = e$ и $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ ».

Описанный алгебраический подход распространяется на типы данных, которые можно аксиоматизировать тождествами или условными тождествами следующим образом.

Определение 2

Тождество – это равенство двух выражений, построенных из имеющихся переменных и символов операций (включая нульместные операции, т.е. константы). Условное тождество – это формула вида «конъюнкция равенств и/или неравенств» \rightarrow «тождество».

Определение 3

Абстрактный тип данных для совокупности (условных) тождеств – это совокупность всех моделей, в которых все эти (условные) тождества имеют место. Реализация абстрактного типа данных, заданного совокупностью (условных) тождеств, – это любая модель, в которой все эти (условные) тождества выполнены.

Неформально говоря, инициальная алгебра для абстрактного типа данных, заданного совокупностью тождеств, – это модель из слов, построенных из символов операций (включая константы), слов, которые более невозможно «упростить» используя эти тождества. При наличии условных тождеств,

инициальная алгебра определяется несколько сложнее, но в любом случае инициальная алгебра сама является реализацией абстрактного типа данных и вкладывается в любую реализацию этого абстрактного типа данных.

Ниже обсуждается алгебраическая семантика упрощённого типа INT языка HeMo, а именно: тип INT без деления и неравенств, который будем обозначать $INT\{/, \leq, \geq, <, >\}$. Алгебраическую семантику этого упрощённого типа можно задать, например, следующей совокупностью тождеств:

- 1) $x+0 = x$,
- 2) $x-x = 0$,
- 3) $x+y = y+x$,
- 4) $(x+y) + z = x + (y+z)$,
- 5) $x-y = x + (-y)$,
- 6) $x \times 0 = 0$,
- 7) $x \times 1 = x$,
- 8) $x \times y = y \times x$,
- 9) $(x \times y) \times z = x \times (y \times z)$,
- 10) $x \times (y+z) = (x \times y) + (x \times z)$,
- 11) $x \times (y-z) = (x \times y) - (x \times z)$,

Тогда абстрактный тип данных $INT\{/, \leq, \geq, <, >\}$ – это совокупность всех моделей, в которых выполнены перечисленные условные тождества, то есть все алгебраические системы, которые называются кольцами, а любое конкретное кольцо (например, поле вычетов по какому-либо простому модулю) – реализация этого абстрактного типа данных $INT\{/, \leq, \geq, <, >\}$.

Наиболее существенно в данном случае, что инициальная модель для абстрактного типа $INT\{/, \leq, \geq, <, >\}$ – это обычные целые числа ... -3, -2, -1, 0, 1, 2, 3, ... с обычными операциями на них. Действительно,

- любое слово, построенное из символов операций «0», «1», «-», «+», «-» и « \times » на основе тождеств 6-11 преобразуется в равное слово, построенное из символов операций «0», «1», «-», «+» и «-» (без « \times »);
- любое слово, построенное из символов операций «0», «1», «-», «+» и «-» на основе тождеств 1-5 преобразуется в равное слово, построенное из одного символа константы «0», нескольких символов константы «1» при помощи нескольких из символов «+» или при помощи нескольких из символов «-»;

- отождествим слово «0» с нулём, всякое слово, построенное из «0» и $k>0$ символов «+» и «1» с числом k , а всякое слово, построенное из «0» и $k>0$ символов «-» и «1» с числом $(-k)$.

Алгебраическую семантику всякого конкретного типа (ТИП1 ARRAY OF ТИП2) можно задать, например, следующей совокупностью условных тождеств:

1. $f = \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x))$,
2. $\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y'), x, y'') = \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y'')$,
3. $x' \neq x'' \rightarrow \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x', y'), x'', y'') = \text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x'', y''), x', y')$,
4. $y = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x, y), x)$,
5. $x' \neq x'' \rightarrow \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(f, x'') = \text{APP}_{[\text{type1} \rightarrow \text{type2}]}(\text{UPD}_{[\text{type1} \rightarrow \text{type2}]}(f, x', y), x'')$.

Соответственно, алгебраическая семантика родового типа ARRAY OF – это порождение новых условных тождеств (таких как только что перечисленные) для каждого вновь образуемого конкретного типа (ТИП1 ARRAY OF ТИП2).

Операционная семантика типов данных языка HeMo

Определение 4

Операционная семантика типов данных – это такая модель аксиоматической семантики или реализация алгебраической семантики, которая с точки зрения человека реализуема на компьютере в терминах очень простых операций над конечными объёмами памяти.

Для языка HeMo операционная семантика типов данных – это модель наследственно-конечных таблиц, которая определяется ниже.

Начнём с операционной семантики типа INT языка HeMo.

Определение. Выберем и зафиксируем систему счисления с некоторым основанием $b>1$ (например, $b=2$ или $b=10$). Тогда значения типа INT – это так называемые «конечные таблицы типа INT». Каждая конечная таблица типа INT - это или последовательность, состоящая из одной цифры «ноль», или любые конечные последовательности цифр фиксированной системы счисления, которая начинается не с цифры «ноль», но, возможно, со знака «-». Каждая такая таблица $(-)a_n \dots a_0$ однозначно определяет математическое целое число $(-1) \times (a_n \cdot b^n + \dots + a_0 \cdot b^0)$. Отношения и операции на таких значениях определяются обычным образом. Например, равенство – это просто совпадение последовательностей, сложение последовательностей без знака – это обычное сложение столбиком, и так далее.

По-видимому, ни у кого не вызывает сомнения, что перед нами пример операционной семантики типа INT. Очевидно, что такая операционная семантика является моделью аксиоматической семантики типа INT и реализует абстрактный тип данных $INT \setminus \{/, \leq, \geq, <, >\}$ ⁵⁴.

Продолжим операционной семантикой типа (ТИП1 ARRAY OF ТИП2) языка HeMo.

Определение. Предположим, что операционная семантика типов ТИП1 и ТИП2 уже определена и для каждого из них значения образуют множества $type1$ и $type2$ конечных таблиц ТИПа1 и ТИПа2 соответственно. Тогда значения типа (ТИП1 ARRAY OF ТИП2) – это произвольные конечные множества пар вида (таблица ТИПа1, таблица ТИПа2). Все такие множества образуют множество $[type1 \rightarrow type2]$ конечных таблиц типа (ТИП1 ARRAY OF ТИП2). Символ константы «new_[type1 → type2]» интерпретируется как пустое множество $\emptyset \in [type1 \rightarrow type2]$. Символ одноместного отношения « $\in [type1 \rightarrow type2]$ » интерпретируется естественным образом. Символ двуместной операции «APP_[type1 → type2]» интерпретируется как операция поиска в конечном множестве $tab \in [type1 \rightarrow type2]$ по значению $arg \in type1$ как по ключу такого $res \in type2$, для которого $(arg, res) \in tab$: если поиск оказывается успешным, то возвращается это значение res ; в противном случае возвращается результат «неопределено». Символ трёхместной операции «UPD_[type1 → type2]» интерпретируется как операция поиска в конечном множестве $tab \in [type1 \rightarrow type2]$ по значению $arg \in type1$ какого-либо $res \in type2$, для которого $(arg, res) \in tab$ и замены пары (arg, res) на новую пару (arg, val) , где $val \in type2$: если поиск оказался успешным, то возвращается результат $(tab \setminus \{(arg, res)\}) \cup \{(arg, val)\}$; в противном случае возвращается результат $tab \cup \{(arg, val)\}$.

По-видимому, ни у кого не вызывает сомнения, что перед нами пример операционной семантики типа (ТИП1 ARRAY OF ТИП2). Очевидно, что такая операционная семантика является моделью аксиоматической семантики типа (ТИП1 ARRAY OF ТИП2) и реализует абстрактный тип данных (ТИП1 ARRAY OF ТИП2) описанные ранее.

⁵⁴ Заметим, что для абстрактного типа данных $INT \setminus \{/, \leq, \geq, <, >\}$ можно определить другую операционную реализацию с «машинным словом фиксированной длины». Для этого надо использовать кольцо вычетов по какому-либо фиксированному целому модулю, все элементы которого однозначно представимы в выбранной системе счисления последовательностями цифр этой системы ограниченной длины и без знака.

Лекция 12. Виртуальная НеМо машина

*Понятие виртуальной НеМо-машины,
её синтаксис и неформальная семантика*

Определение 5

Виртуальная машина – это гипотетический компьютер, а язык виртуальной машины – это ассемблер виртуальной машины, то есть язык программирования в системе команд виртуальной машины. Предполагается, что виртуальная машина способна непосредственно исполнить или (как принято говорить) «интерпретировать» любую команду языка виртуальной машины⁵⁵. Реализовать виртуальную машину или интерпретатор языка виртуальной машины на конкретной машине в конкретной операционной системе – это означает написать программу для этой конкретной обстановки, которая симулирует работу гипотетической виртуальной машины по интерпретации команд языка виртуальной машины.

Язык виртуальной НеМо-машины состоит из программ, синтаксис и семантика которых определяются ниже, а прагматика – в следующих трех лекциях 13-15. Неформально говоря, язык виртуальной НеМо машины – это Basic с недетерминированными переходами, семантика программ на этом языке – это всевозможные варианты недетерминированных вычислений, начатые в каком-либо начальном состоянии памяти и представленные в виде дерева вычислений, в котором ветвление происходит там, где происходит недетерминированный выбор следующего оператора. Следовательно, реализация виртуальной НеМо машины или интерпретатор для языка виртуальной НеМо машины – это программа для конкретной языковой и операционной обстановки (например, программа на C++ для Visual Studio 6.0 под Windows2000), которая выполняет пошаговое построение (или обход) такого дерева по любому заданному начальному состоянию и любой заданной программе виртуальной НеМо машины.

Определение. Каждая программа виртуальной НеМо машины состоит из двух частей: конечного множества описаний переменных и конечного множества помеченных операторов⁵⁶. Каждое описание само по себе имеет такой же вид, как и в языке НеМо, а вся совокупность описаний перемен-

⁵⁵ Например, язык виртуальной JAVA машины – это ассемблер виртуальной JAVA машины, а виртуальная JAVA машина – это интерпретатор команд языка виртуальной JAVA машины.

⁵⁶ Таким образом, командами виртуальной НеМо машины являются описания переменных и помеченные операторы.

ных какой-либо программы виртуальной HeMo машины удовлетворяет тем же ограничениям, что и совокупность описаний в HeMo программах, а именно: каждая используемая переменная должна быть описана, а каждая описанная переменная не может иметь более одного описания в программе. Множество помеченных операторов программы образует её тело. В качестве меток в теле программ используются целые десятичные числа без знака. Каждая метка может метить несколько (но сколько угодно) операторов в теле программы⁵⁷.

Определение. Помеченные операторы бывают двух видов: операторы присваивания и условные операторы. Всякий помеченный оператор присваивания имеет следующий вид

⟨метка⟩ : ⟨переменная⟩ := ⟨выражение⟩ goto ⟨множество_меток⟩;

а всякий помеченный условный оператор имеет следующий вид

⟨метка⟩ : ⟨условие⟩ then ⟨множество_меток⟩ else ⟨множество_меток⟩;

где ⟨выражение⟩ и ⟨условие⟩ такие же, как в языке HeMo, а в качестве ⟨множеств_меток⟩ могут использоваться как пустое, так и любое одноэлементное, а так же и произвольное многоэлементное конечное множество меток⁵⁸.

Сначала поясним семантику языка виртуальной HeMo-машины неформально. Каждая программа начинает выполняться с отведения памяти под описанные переменные в соответствии с их описаниями. Первый исполняемый оператор любой программы – любой оператор тела, помеченный меткой «0». Исполнение (или интерпретация) оператора присваивания «k: x:=t goto L» из тела программы состоит в том, что новое значение переменной «x» становится равным прежнему значению выражения «t», после чего управление передаётся какому-либо оператору, помеченному какой-либо меткой из множества «L»⁵⁹. Исполнение (или интерпретация) условного оператора «k: φ then L+ else L-» из тела программы состоит в том, что управление передаётся какому-либо оператору, помеченному какой-либо меткой из множества «L+», если φ имеет место (выполнено, верно), и помеченному какой-либо меткой из множества «L-» в противном случае⁴. Завершается

⁵⁷ Эта возможность соответствует недетерминированному выбору из операторов, помеченных одинаковыми метками.

⁵⁸ Пустое множество соответствует исключительной ситуации или аварииной остановке, многоэлементное множество соответствует недетерминированному выбору одной из меток, а одноэлементное множество – псевдодетерминизму (т. к. любая метка может метить несколько операторов).

⁵⁹ Если множество меток пусто, то (как уже говорилось) возникает исключительная ситуация аварииной остановки.

исполнение программы передачей управления какой-либо метке, которая не метит ни одного оператора в теле программы.

Вычислительная операционная семантика

Для удобства формального определения вычислительной операционной семантики языка виртуальной НеМо-машины выберем произвольно и зафиксируем для определённости программу α виртуальной НеМо машины. Тем самым фиксировано множество описаний переменных этой программы и множество операторов.

Определение 6

Состояние (памяти программы α) – это произвольное отображение, которое сопоставляет каждой описанной переменной некоторое значение (возможно, неопределённое) в соответствии с операционной семантикой её типа (то есть некоторую таблицу). Множество всех состояний программы α называется её пространством состояний и обозначается $SP(\alpha)$.

Определение. Каждое состояние $s \in SP(\alpha)$ позволяет приписать значение $s(t)$ каждому синтаксически правильному выражению t языка виртуальной НеМо машины, построенному только из описанных переменных, следующим образом:

- если t – какая-либо переменная, то $s(t)$ определено так как s – состояние;
- если t имеет вид $f(t_1, \dots, t_n)$, где $n \geq 0$ – число формальных аргументов префиксного символа f операции F , а t_1, \dots, t_n – выражения, которые являются фактическими аргументами, то $s(t) = F(s(t_1), \dots, s(t_n))$, если все значения $s(t_1), \dots, s(t_n)$ определены, и $s(t) =$ неопределено в противном случае;
- если t имеет вид $(t_1 f \dots t_n)$, где $n \geq 0$ – число формальных аргументов инфиксного символа f операции F , а t_1, \dots, t_n – выражения, которые являются фактическими аргументами, то $s(t) = F(s(t_1), \dots, s(t_n))$, если все значения $s(t_1), \dots, s(t_n)$ определены, и $s(t) =$ неопределено в противном случае;
- если t имеет вид $(t_1 \dots t_n) f$, где $n \geq 0$ – число формальных аргументов суффиксного символа f операции F , а t_1, \dots, t_n – выражения, которые являются фактическими аргументами, то $s(t) = F(s(t_1), \dots, s(t_n))$, если все значения $s(t_1), \dots, s(t_n)$ определены, и $s(t) =$ неопределено в противном случае.

Определение. Конфигурация (программы α) – это произвольная пара вида (l, s) , где l – метка, которая встречается в программе α , а s – состояние программы α . Конфигурация называется начальной, если её метка есть $\langle 0 \rangle$.

Конфигурация называется заключительной, если её метка не метит ни одного оператора (в программе α). Аварийная (или исключительная) конфигурация (программы α) – это зарезервированное слово «авост».

Определение. Срабатывание (в программе α) помеченного оператора присваивания «l' x:= t goto L» (где «x», «t» и «L» – произвольные переменная, выражение и множество меток) – это любая пара конфигураций (l' , s') (l'' , s'') программы α такая, что $l'' \in L$, а $s' = \text{UPD}(s'', x, s'(t))$, т.е. s'' отличается от s' только значением переменной x , которое в s'' равно $s'(t)$.

Определение. Срабатывание (в программе α) помеченного условного оператора «l': if ϕ then $L+$ else $L-$ » (где « ϕ », « $L+$ » и « $L-$ » – произвольное условие и пара множеств меток) – это любая пара конфигураций (l' , s') (l'' , s'') программы α , для которой выполнено одно из двух:

- $l'' \in L+$, $s'' = s'$ и $s' \models \phi$, т.е. в s' имеет место ϕ ;
- $l'' \in L-$, $s'' = s'$ и неверно $s' \models \phi$, т.е. в s' не имеет места ϕ .

Определение. Срабатывание (программы α) – это произвольная пара конфигураций, которая является срабатывание какого-либо оператора (присваивания или условного) в этой программе.

Определение. Аварийное срабатывание (программы α) – это любая пара конфигураций (l , s) (авост) программы α , для которой выполнено одно из трёх:

- в теле программы есть помеченный оператор присваивания «l: x:= t goto \emptyset », где «x» и «t» – произвольные переменная и выражение;
- в теле программы есть помеченный условный оператор «l: if ϕ then \emptyset else L », где « ϕ » и « L » – произвольное условие и множество меток, такой, что $s \models \phi$, т.е. в s имеет место ϕ ;
- в теле программы есть помеченный условный оператор «l: if ϕ then L else \emptyset », где « ϕ » и « L » – произвольное условие и множество меток, такой, что неверно $s \models \phi$, т.е. в s не имеет места ϕ .

Определение. Трасса (программы α) – это произвольная конечная или бесконечная последовательность конфигураций (включая, возможно, аварийную конфигурацию «авост») $\text{cnf}_0 \dots \text{cnf}_i, \dots$ такая, что любая пара соседних конфигураций $\text{cnf}_i \text{ cnf}_{i+1}$ из этой последовательности является срабатыванием программы α (возможно, аварийным). Трасса называется началь-

ной, если она начинается с начальной конфигурации. Трасса называется финальной, если она бесконечна или если она заканчивается заключительной конфигурацией. Трасса называется аварийной, если она заканчивается аварийной конфигурацией. Трасса называется полной, если она является начальной и финальной одновременно.

Определение. Вычислительная семантика программы α - это множество $[\alpha]$ всех таких пар состояний (s', s'') , для которых существует конечная полная трасса, начинающаяся (в конфигурации c) состоянием s' , а заканчивающаяся (в конфигурации c) состоянием s'' .

Семантические деревья

Для определённости (как и в предыдущем пункте) выберем произвольно и зафиксируем программу α виртуальной HeMo машины. Тем самым фиксировано множество состояний и конфигураций.

Множество всех полных конечных трасс программы α , которые начинаются в каком-либо фиксированном состоянии $s \in SP(\alpha)$ можно представить в виде так называемого семантического дерева $TR_\alpha(s)$, «склеив» общие префиксы полных трасс.

Определение дерева $TR_\alpha(s)$:

- вершины дерева $TR_\alpha(s)$ – это конечные последовательности конфигураций, которые начинаются в конфигурации c состоянием s и являются префиксами полных трасс;
- для любых двух вершин $cnf_0' \dots cnf_i'$ и $cnf_0'' \dots cnf_j''$ дерева $TR_\alpha(s)$, последовательность $cnf_0'' \dots cnf_j''$ является наследником последовательности $cnf_0' \dots cnf_i'$ тогда и только тогда, когда $j=(i+1)$ и $cnf_0'=cnf_0''$, ... $cnf_i'=cnf_i''$.

Очевидно, что для любых состояний $s', s'' \in SP(\alpha)$ имеет место следующее: $(s', s'') \in [\alpha]$ тогда и только тогда, когда в $TR_\alpha(s')$ есть лист с состоянием s'' .

Множество всех полных и аварийных трасс программы α , которые начинаются в каком-либо фиксированном состоянии $s \in SP(\alpha)$ тоже можно представить в виде так называемого обобщённого (или расширенного) семантического дерева $ET_\alpha(s)$, «склеив» общие префиксы трасс (как полных, так и аварийных).

Определение расширенного дерева $ET_\alpha(s)$:

- вершины дерева $ET_{\alpha}(s)$ – это конечные последовательности конфигураций, которые начинаются в конфигурации с состоянием s и являются префиксами полных или аварийных трасс;
- для любых двух вершин $cnf_0' \dots cnf_i'$ и $cnf_0'' \dots cnf_j''$ дерева $ET_{\alpha}(s)$, последовательность $cnf_0' \dots cnf_j''$ является наследником последовательности $cnf_0' \dots cnf_i'$ тогда и только тогда, когда $j=(i+1)$ и $cnf_0'=cnf_0'', \dots, cnf_i'=cnf_i''$.

Опять очевидно, что для любых состояний $s', s'' \in SP(\alpha)$ имеет место следующее: $(s', s'') \in [\alpha]$ тогда и только тогда, когда в $TR_{\alpha}(s')$ есть лист с состоянием s'' .

Но данные выше определения деревьев $TR_{\alpha}(s)$ и $ET_{\alpha}(s)$ не являются конструктивными, так как они неявно предполагают, что множество префиксов трасс уже дано, а надо только его представить в виде дерева. Однако $ET_{\alpha}(s)$ можно определить и другим, более конструктивным способом, если в каждой вершине расширенного дерева $ET_{\alpha}(s)$ вместо всей последовательности конфигураций оставлять только последнюю конфигурацию.

Конструктивное определение расширенного дерева $ET_{\alpha}(s)$:

- корень дерева $ET_{\alpha}(s)$ – это начальная конфигурация $(0, s)$;
- для любой вершины cnf' дерева $ET_{\alpha}(s)$, её наследниками являются вершины помеченные конфигурациями cnf'' такими, что (cnf', cnf'') является срабатыванием (возможно, аварийным) программы α .

Пример.

Пусть α следующая программа виртуальной HeMo машины:

VAR $x : INT$; VAR $y : INT$;

0 : if $x < 0$ then \emptyset else $\{1\}$

1 : $y := y + 1$ goto $\{1, 2\}$

1 : $y := y - 1$ goto $\{1, 2\}$

2 : if $(y + y \leq x) \ \& \ (y + y + 1 > x)$ then $\{3\}$ else \emptyset

Эта простая программа для неотрицательных значений переменной x вычисляет в переменной y частное $[x/2]$ от деления x нацело на 2, а при отрицательных значениях переменной x совершает аварийную остановку.

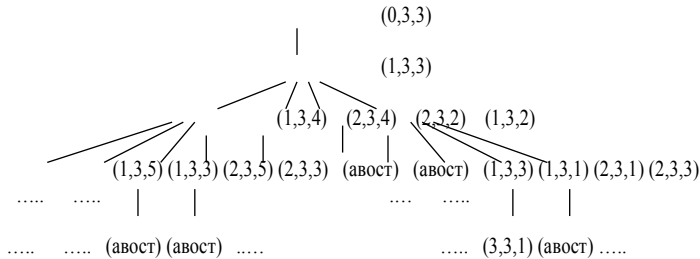
Состояния такой программы α удобно представлять парами вида («значение x », «значение y »), а конфигурации - тройками вида («метка», «значение x », «значение y »). Вот несколько примеров полных и аварийных трасс, начинающихся с состояния $(3, 3)$:

$(0, 3, 3) (1, 3, 3) (1, 3, 4) (1, 3, 5) (1, 3, 6) (1, 3, 7) (1, 3, 8) (1, 3, 9) \dots$

$(0, 3, 3) (1, 3, 3) (1, 3, 2) (1, 3, 1) (2, 3, 1) (3, 3, 1)$

(0,3,3) (1,3,3) (1,3,2) (2,3,3) (авост)

Верхние четыре этажа соответствующего расширенного дерева вычислений выглядит примерно следующим образом:



Вернёмся к анализу общего случая. Так как расширенное семантическое дерево отличается от простого семантического дерева только (возможно) бесконечными полными и аварийными трассами, то для любых состояний $s', s'' \in SP(\alpha)$ имеет место следующее: $(s', s'') \in [\alpha]$ тогда и только тогда, когда в $ET_\alpha(s')$ есть лист с состоянием s'' . Очевидно также, что конструктивное определение расширенного семантического дерева позволяет эффективно перечислить все состояния, которые встречаются в его листьях. Поэтому проблема перечисления по произвольному входному состоянию s' всех выходных состояний s'' таких, что $(s', s'') \in [\alpha]$, является частично разрешимой (рекурсивно перечислимой).

Корректный метод обхода расширенных семантических деревьев

Рассмотрим подробнее, как именно можно обойти расширенное семантическое дерево. Заметим, что, вообще говоря, расширенное семантическое дерево имеет динамическую, а не статическую природу, т.к. оно строится во время исполнения программы. В лекции 6 мы обсуждали два метода обхода (виртуальных) деревьев:

- метод отката,
- метод ветвей и границ.

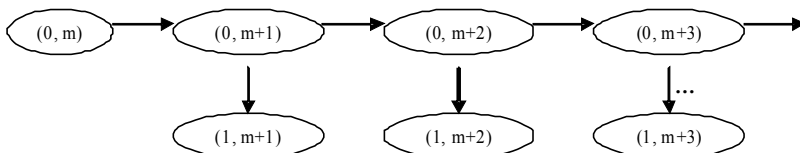
Попытаемся применить оба этих алгоритма для обхода расширенного семантического дерева.

Отметим, что обход расширенных семантических деревьев методом отката в общем случае не применим, так как легко может заикливаться без каких-либо результатов. Для этого рассмотрим следующую тривиальную

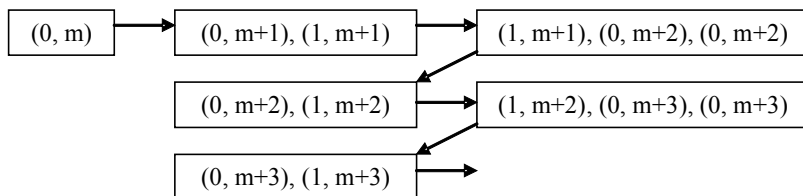
программу α виртуальной HeMo-машины, состоящую из единственного оператора присваивания:

VAR m : INT; 0: $m := m+1$ goto {0, 1} .

Для любого начального значения m переменная m расширенное дерево вычислений по этой программе $ET_\alpha(s: m \rightarrow m)$ имеет следующий вид:



Обход методом отката в данном случае может вылиться в бесконечный «спуск» по бесконечной ветке без единого результата. Наоборот, обход этого же дерева методом ветвей и границ сведётся к обходу в ширину и будет результативным, а именно: динамика очереди будет, например, следующая:



(то есть на каждом втором шаге будет получаться новая заключительная конфигурация, и последовательность получающихся заключительных конфигураций будет $(1, m+1)$, $(1, m+2)$, $(1, m+3)$ и так далее).

Для того, что бы адаптировать метод ветвей и границ для обхода расширенных семантических деревьев надо ответить на вопрос, каково целевое условие и граничное условие? В качестве целевого условия мы примем следующее естественное условие: «Лист является заключительной конфигурацией, которая ещё не посещалась во время обхода». Тогда в качестве граничного условия В имеет смысл принять следующее: «Верши является конфигурацией, которая уже посещалась во время обхода». Поэтому приведённая в лекции 6 общая схема метода ветвей и границ преобразуется в следующий алгоритм обхода расширенных семантических деревьев.

рых конфигураций, которые встречаются в $ET_\alpha(s)$ на высоте $height(curn)$;

- 1.2) очередь QUE не имеет повторов и состоит из некоторых конфигураций, которые не входят в VIS , но встречаются в $ET_\alpha(s)$ на высотах $height(curn)$ или $(height(curn) + 1)$;
 - 1.3) для любой пары конфигураций cnf' и cnf'' программы α , если $cnf' \in VIS$ а cnf'' является непосредственным наследником cnf' в $ET_\alpha(s)$, то $cnf'' \in VIS$ или $cnf'' \in QUE$, а в противном случае $cnf'' = curn$.
2. Для любой конфигурации $cnf \in ET_\alpha(s)$ наступит такой момент исполнения алгоритма, в который $curn = cnf$.

Доказательство. Применим принципы доказательства З. Манни и А. Пнуели для условий безопасности и прогресса. Сначала докажем условие безопасности, а потом – условие прогресса.

Условие безопасности.

В начальный момент исполнения алгоритма условие безопасности имеет место очевидным образом. Теперь предположим, что в некоторый момент исполнения M алгоритма или этот момент является начальным, или выполнены условия 1.1-1.3, и докажем, что в непосредственно следующий момент N эти условия тоже будут выполнены. Разберём возможные случаи для M и N .

- Если M и N – это моменты исполнения до и после оператора присваивания « $curn := head(QUE)$ », то момент M или вообще начальный момент, или получался после завершения предыдущей итерации цикла. В первом из этих случаев в момент M имеем $QUE = \langle(0, s)\rangle$, $VIS = \emptyset$, а в момент N имеем $curn = (0, s)$, $height(curn) = 0$ и по-прежнему $QUE = \langle(0, s)\rangle$, $VIS = \emptyset$, то есть условия 1.1-1.3 выполнены в момент N очевидным образом. Во втором из этих случаев момент M не является начальным, для него выполнены условия 1.1-1.3, а N отличается от M только значением переменной $curn = head(QUE)$; но M получается после выполнения оператора присваивания « $VIS := VIS \cup \{curn\}$ », поэтому значение переменной $curn$ в момент M не потеряно, а включено в множество VIS ; следовательно, условия 1.1-1.3 имеют место и в момент времени N .
- Если M и N – это моменты исполнения до и после оператора присваивания « $QUE := tail(QUE)$ », то момент M не является начальным и для него выполнены условия 1.1-1.3. Момент N отличается от M только значением переменной $QUE = tail(QUE)$. Но M получается после выпол-

нения оператора присваивания « $curn = head(QUE)$ », поэтому конфигурация, хранившаяся в голове QUE в момент M не потеряно, а сохранено в переменной $curn$; следовательно, условия 1.1-1.3 имеют место и в момент времени N .

- Если M и N – это моменты исполнения до и после оператора присваивания « $QUE := QUE \wedge next$ », то истинность условий 1.1 и 1.3 в момент M влечёт их истинность в момент N , т.к. значение переменных VIS и $curn$ не изменяется, а переменной QUE – расширяется за счёт вершины $next$. Т.к. присваивание « $QUE := QUE \wedge next$ » выполняется только в том случае, когда $next \notin VIS$, $next \notin QUE$, $next \neq curn$ и $(curn, next)$ – срабатывание α , то после этого присваивания QUE не приобретёт повторов или копий конфигураций из VIS , но расширится за счёт вершины $next$, которая встречается в $ET_\alpha(s)$ на высоте $height(next) = (height(curn) + 1)$; поэтому, если условие 1.2 было выполнено в момент M , то оно будет выполнено и в момент N .
- Если M и N – это моменты исполнения до и после оператора присваивания « $VIS := VIS \cup \{curn\}$ », то истинность условия 1.2 в момент M влечёт его истинность в момент N , т.к. значение переменной QUE не изменяется. Истинность условия 1.1 также сохраняется при переходе от M к N , т.к. значение VIS изменяется только на $\{curn\}$. Нам остаётся показать, что условие 1.3 тоже сохраняется при переходе от момента M к моменту N . Пусть условие 1.3 выполнено в момент M и пусть cnf' и cnf'' – произвольная пара конфигураций программы α такая, что cnf'' – наследник cnf' в $ET_\alpha(s)$. Если $cnf' \in VIS$ в момент N , то $cnf' \in VIS$ в момент M или $cnf' = curn$. В первом из этих случаев $cnf'' \in VIS$, или $cnf'' \in QUE$, или $cnf'' = curn$ в момент N , т.к. множество VIS только расширилось, а очередь QUE не изменилась. Во втором случае (т.е. когда $cnf' = curn$) имеем: в момент M все наследники cnf' уже были добавлены в QUE , т.к. присваиванию « $VIS := VIS \cup \{curn\}$ » непосредственно предшествует условный оператор

IF $curn$ – заключительная конфигурация α THEN output ($curn$)
ELSE

FOR EACH { $next : next \notin VIS, next \notin QUE, next \neq curn$ и $(curn, next)$ срабатывание α } DO $QUE := QUE \wedge next$.

Таким образом, доказательство условия безопасности завершено.

Условие прогресса.

Пусть cnf – произвольная конфигурация из $\text{ET}_\alpha(s)$, N – минимальное расстояние от cnf до корня в $\text{ET}_\alpha(s)$, а N_H – это число всех конфигураций из $\text{ET}_\alpha(s)$, высота которых не больше $(N+1)$. Примем в качестве фундированного множества (D, \leq) множество натуральных чисел с естественным отношением порядка, а в качестве отображения $f: \text{ET}_\alpha(s) \rightarrow D$ следующую функцию:

$$F(\text{конфигурация}) = \begin{cases} N_H - |\text{VIS}|, & \text{если } \text{cnf} \notin \text{VIS}, \\ 0 & \text{в противном случае.} \end{cases}$$

Очевидным образом все операторы присваивания алгоритма не увеличивают значения этой функции, а один из этих операторов ($\langle \text{VIS} := \text{VIS} \cup \{\text{cnp}\} \rangle$) – даже уменьшает, если до сих пор конфигурация cnf всё ещё не попала в VIS . Следовательно, рано или поздно cnf попадает в VIS . Но единственный способ попасть в VIS – это через присваивание $\langle \text{VIS} := \text{VIS} \cup \{\text{cnp}\} \rangle$ в момент когда $\text{cnp} = \text{cnf}$. Таким образом, условие прогресса тоже доказано. ■

Лабораторная работа 3. Интерпретация языка виртуальной НеМо-машины

Цель лабораторной работы: ознакомление на практике с понятием виртуальной машины и интерпретацией программ.

Что входит в лабораторную работу:

1. Формальное определение регулярного синтаксиса ВАШЕГО диалекта языка виртуальной НеМо-машины в нотации Бэкуса-Наура или в форме синтаксических диаграмм.
2. Интерпретатор для программ на ВАШЕМ диалекте языка виртуальной НеМо-машины.

Этапы:

1. Разработка и формализация (в нотации Бэкуса-Наура или синтаксических диаграмм) синтаксиса ВАШЕГО диалекта языка виртуальной НеМо-машины.
2. Разработка алгоритма интерпретации ВАШЕГО диалекта языка виртуальной НеМо машины.
3. Дизайн и реализация программы-интерпретатора ВАШЕГО диалекта языка виртуальной НеМо-машины.

- Требования «заказчика» к диалекту языка виртуальной НеМо-машины.
1. Полная синтаксическая совместимость по типам данных с принятым ВАМИ диалектом НеМо.
 2. Семантика типа INT – кольцо вычетов по модулю некоторого параметра MAXINT, а типа (INT ARRAY OF INT) – наследственно –конечные таблицы с индексами и значениями из этого кольца.
 3. Поддержка выражений с одним и/или двумя операндами в правых частях операторов присваивания и в условиях тестов.
 4. Поддержка отношений «=», «<», «>» «<=» и «>=» без булевских операций.
 5. Поддержка числа переменных, не меньше чем число переменных принятого ВАМИ диалекта НеМо.
 6. Поддержка любых конечных множеств в полях «goto», «then» и «else».

Требований «заказчика» к представлению алгоритма интерпретации:

Алгоритм должен быть представлен в форме псевдокода или блок-схемы высокого уровня и небольшой сопроводительной записки-комментария об используемых структурах данных, операциях на них и т.п. Общий объём описания алгоритма и сопроводительной записки – 1-2 стр. (Блок-схема может быть нарисована от руки.)

Требования «заказчика» к программе-интерпретатору:

1. Предусловие: Входные данные – ASCII файл с расширением VNM.
2. Условия живости:
 - 2.1. 1-е условие живости: если входной файл не является программой на принятом диалекте языка виртуальной НеМо-машины, то на экран выдаётся сообщение о «первой» синтаксической ошибке (без диагностики).
 - 2.2. 2-е условие живости: если входной файл является программой на принятом диалекте языка виртуальной НеМо-машины, то интерпретатор начинает с запроса о значении параметра MAXINT и запросов о значениях переменных в начальном состоянии s_{ini} ; каждый запрос имеет форму «переменная»«?», в ответ на каждый такой запрос пользователь вводит десятичное целое число (возможно со знаком).
 - 2.3. 3-е условие живости: если входной файл является программой на принятом диалекте языка виртуальной НеМо-машины, то для любого листа ($_ , s_{fin}$) расширенного семантического дерева с корнем

$(0, s_{ini})$ интерпретатор во время работы выведет на экран состояние s_{fin} в отдельной строчке перечислением значений всех переменных в этом состоянии в виде $\langle \text{переменная} \rangle \langle : \rangle \langle \text{значение} \rangle$.

- 2.4. 4-е условие живости: если входной файл является программой на принятом диалекте языка виртуальной НеМо-машины, и множество всех вершин расширенного семантического дерева с корнем $(0, s_{ini})$ является конечным, то интерпретатор завершает свою работу выводом на экран слова «Всё!».
3. Условия безопасности: если входной файл является программой на принятом диалекте языка виртуальной НеМо-машины, то
 - 3.1. Для всякого выведенного на экран состояния s_{fin} в расширенном семантическом дереве с корнем $(0, s_{ini})$ есть лист $(_, s_{fin})$.
 - 3.2. Последовательность выведенных на экран состояний не содержит повторений.

Несколько примеров программ на некоем диалекте языка Виртуальной НеМо-машины. Во всех трёх примерах все используемые переменные имеют тип INT, во всех этих примерах вычисляется заключительное значение переменной m , равное целой части корня из начального значения переменной n , т.е. $s_{fin}(m) = \lfloor (s_{ini}(n))^{1/2} \rfloor$ (при условии, что $(\lfloor (s_{ini}(n))^{1/2} \rfloor + 1)^2 \leq \text{MAXINT}$).

В первом примере программа хаотически вычисляет («угадывает») значение m переменной m (метки 0 и 10), а затем сравнивает m^2 и $(m+1)^2$ со значением n переменной n (метки 21 и 30). Выходом является метка 12345. Программа завершается аварийной остановкой, если значение угаданное значение слишком велико (метка 21), или если это значение слишком мало (метка 30).

```

0: m:=0 goto {10}
10: m:=m+1 goto {10, 15}
10: m:=m-1 goto {10, 15}
15: if m>=0 then {20} else {15}
20: y:=m*m goto {21}
21: if y<=n then {22} else {}
22: y:=y+m goto {23}
23: y:=y+m goto {24}
24: y:=y+1 goto {30}
30: if y>>n then {12345} else {}

```

Во втором примере вычисляются последовательные значения $m = 1, 2, \dots$ пока m не превзойдёт n (метки 10 – 22). Завершается программа вычислением значения $(m-1)$ и выходом по метке 5.


```

0: if n>=0 then{10} else{0}
10: m:=0 goto{11}
11: y:=0 goto{20}
20: if y>>n then{30} else{21}
21: m:=m+1 goto{22}
22: y:=m*m goto{20}
30: m:=m-1 goto{5}

```

Третья программа просто оптимизирует вычисления значения $m2$.

```

0: if n>=0 then{10} else{}
10: m:=0 goto{11}
11: y:=0 goto{20}
20: if y>>n then{30} else{21}
21: y:=y+m goto{22}
22: y:=y+m goto{23}
23: y:=y+1 goto{24}
24: m:=m+1 goto{20}
30: m:=m-1 goto{5}

```

Лекция 13. Введение в семантику программ (на примере языка НеМо)

Ещё раз о неформальной семантике НеМо-программ

Каждая программа на языке НеМо состоит из описаний и тела. Поэтому семантика программ на языке НеМо состоит из двух слагаемых: семантики описаний и семантики тела программы. Однако, прежде чем перейти к формализации семантики описаний и тела программы, обсудим зачем нужна формальная семантика программ и какие виды семантики существуют, напомним и конкретизируем неформальную семантику вычислительных программ на языке НеМо.

Зачем нужна формальная семантика программ, и какие виды семантики существуют? Здесь дело обстоит примерно так же, как с семантикой типов данных: существует операционная, аксиоматическая и математическая семантика программ. Операционная семантика программ нужна для того, чтобы описать, как произвести вычисления, который заданы программой, причём так, что эти вычисления мог выполнить неодоушевлённый исполнитель — компьютер. Аксиоматическая семантика нужна для того, что бы «рассуждать» на чисто синтаксическом уровне о свойствах вычислений, заданных

программой, причём так, что провести эти рассуждения может тот же компьютер. Наоборот, математическая семантика программ носит человеко-ориентированный характер, имеет некий метауровень, на котором человек может доказывать свойства целых классов программ.

Теперь обсудим неформальную семантику описаний в программе. Описания определяют типы переменных, то есть (в первом приближении) множества возможных значений каждой из переменных программы. Если для каждой переменной в программе на языке НеМо известно множество её возможных значений, то появляется возможность организовать размещение, хранение и доступ (для использования и/или изменения) к текущим значениям этой переменной во время вычислений по программе. В рамках операционной семантики типов данных в языке НеМо мы приняли модель наследственно-конечных таблиц, согласно которой любое значение любого типа данных представимо в виде таблицы конечного размера. Такую конечную таблицу можно хранить механическим, электронным, оптическим или ещё каким-либо образом; соответственно методам хранения должны быть предусмотрены и методы доступа к таким таблицам. Например, можно предложить следующий механический метод хранения в виде записи на листочках тетради. В таком случае доступ для использования – это просто чтение с последнего заполненного листочка, а доступ для изменения – это запись на новый чистый листочек. Для языка НеМо неважно, как организовано хранение и доступ к значениям каждой отдельной переменной программы: одни переменные могут храниться механическим образом, другие – электронным, третьи – оптическим и так далее. Для программы на языке НеМо важно только то, что для каждой её переменной каким-то образом выбран и фиксирован на время выполнения этой программы адекватный метод хранения и доступа к текущим значениям этой переменной. Организация памяти – это набор имеющихся средств хранения и доступа для значений разных типов, распределение памяти программы на языке НеМо – это как раз выбор и фиксация на время выполнения этой программы метода хранения и доступа для каждой переменной этой программы, а состояние памяти программы – это совокупность текущих значений всех переменных программы в какой-либо момент её исполнения, представленных в форме, соответствующей распределению памяти этой программы (например, для переменной, хранимой механическим образом в виде записи в тетрадке, – листочек с последней на этот момент записью, для переменной, хранимой электронным образом в виде напряжения между анодом и катодом, – напряжение в этот же момент времени и так далее).

Если абстрагироваться от конкретной организации и распределения памяти программы, то мы естественно приходим к понятию состояния про-

граммы (которое уже было введено в предыдущей лекции 12 в контексте языка виртуальной НеМо-машины⁶⁰): состояние – это совокупность текущих значений всех переменных программы в какое-либо мгновение её исполнения, представленных уже в абстрактной (математической) форме (то есть целых чисел для переменных типа INT, частичных функций с конечной областью определения – для массивов).

И, наконец, обсудим неформальную семантику тела программы, причём сосредоточимся на вычислительных программах, то есть программах, которые за конечное время преобразует начальные (входные) в заключительные (выходные) значения. Преобразования значений переменных происходит в теле программы. Выше мы уже условились называть состоянием программы мгновенный набор значений всех переменных программы. Значит, для вычислительной программы семантика её тела – это отношение вход-выход на состояниях программы, какой набор входных значений переменных может быть преобразован телом программы, в какой выходной набор значений переменных. Тело любой программы состоит из присваиваний и тестов (проверок), «соединённых» при помощи конструкторов последовательного исполнения «;», недетерминированного выбора « \cup » и недетерминированной итерации «*». Поэтому ниже мы разберём по порядку начиная с присваиваний и тестов на неформальном уровне это отношение вход-выход для вариантов тела программы при фиксированных описаниях переменных.

Как уже говорилось, смысл (т.е. неформальная семантика) оператора присваивания – изменение значения переменной. Это новое значение замещает старое значение, которое переменная имела непосредственно перед этим присваиванием. Но старое значение переменной – часть «старого» или входного состояния перед выполнением оператора присваивания, а новое значение – часть «нового» или выходного состояния. Значит, отношение вход-выход для тела программы, состоящего из одного оператора присваивания – это следующее отношение: выходное состояние отличается от входного только тем, что значение переменной, которой производилось присваивание, изменилось на значение, равное правой части этого присваивания.

Смысл теста состоит в том, что когда при проверке условие теста оказывается истинно (верно), то ничего не происходит; в противном случае тест аннулирует вычисление, которое привело к этой проверке. Значит с телом программы, которое состоит из единственного теста, можно связать следующее отношение вход-выход на состояниях программы: входное и выходное состояния совпадают в том и только том случае, когда условие

⁶⁰ См. определение 2.

теста верно; в случае, когда условие не верно, входному состоянию не соответствует никакое выходное состояние.

Интуитивно $(\alpha ; \beta)$ соответствует последовательному исполнению сначала α , потом - β , т.е. следующей последовательности действий:

- поступает входное состояние для $(\alpha ; \beta)$,
- α преобразует его в своё выходное состояние,
- β преобразует выходное состояние α в своё выходное состояние,
- $(\alpha ; \beta)$ выдаёт выходное состояние β .

Таким образом, отношение вход-выход для тела $(\alpha ; \beta)$ можно переформулировать следующим образом: существует некоторое промежуточное состояние такое, что

- входное состояние и это промежуточное состояние являются входом-выходом α ,
- это промежуточное состояние и выходное состояние являются входом-выходом β .

Интуитивно $(\alpha \cup \beta)$ соответствует выбору одного из возможных исполнений α или β . Поэтому отношение вход-выход для тела $(\alpha \cup \beta)$ можно переформулировать следующим образом: входное и выходное состояния

- или являются входом-выходом α ,
- или являются входом-выходом β .

Интуитивно $(\alpha)^*$ соответствует исполнению α или 0-раз, или 1-раз, или 2-раза, и т.д., т.е. недетерминированному выбору между следующими числительными альтернативами:

- не предпринимать никаких действий (т.е. выполнить α ноль раз),
- выполнить α (т.е. выполнить α один раз),
- выполнить $(\alpha ; \alpha)$ (т.е. выполнить α последовательно 2 раз),
- выполнить $((\alpha ; \alpha) ; \alpha)$ (т.е. выполнить α последовательно 3 раз),
- и т.д.

Поэтому отношение вход-выход для тела $(\alpha)^*$ можно переформулировать следующим образом: существует такая конечная последовательность состояний, которая начинается с входного состояния, заканчивается выходным состоянием, а каждое из состояний в этой последовательности связано с непосредственно следующим за ним состоянием отношением вход-выход для тела α .

Операционная семантика описаний

Выберем произвольно и зафиксируем до конца лекции совокупность описаний переменных, в которой каждая описанная переменная имеет

единственное описание. Мы также будем предполагать, что фиксированная совокупность описаний описывает все переменные, которые будут встречаться в телах программ вплоть до конца этой лекции.

Состояние – это произвольное отображение, которое сопоставляет каждой описанной переменной некоторое значение в соответствии с операционной семантикой её типа (т.е. некоторую конечную таблицу). Множество всех состояний называется пространством состояний. Вплоть до конца этой лекции мы будем обозначать через SP множество всех состояний для выбранной выше и фиксированной совокупности описаний. Каждое состояние s позволяет приписать значение $s(t)$ каждому синтаксически правильному выражению t языка НеМо, построенному только из описанных переменных, так же, как это уже было определено в предыдущей лекции 12:

- если t – какая-либо переменная, то $s(t)$ определено т.к. s – состояние;
- если t имеет вид $f(t_1, \dots, t_n)$, или $(t_1 \ f \ \dots \ t_n)$, или $(t_1, \dots, t_n)f$ где $n \geq 0$ – число формальных аргументов символа f , F – семантика символа операции, а t_1, \dots, t_n – выражения, которые являются фактическими аргументами, то $s(t) = F(s(t_1), \dots, s(t_n))$, если все значения $s(t_1), \dots, s(t_n)$ определены, и $s(t) =$ неопределено в противном случае.

Традиционная операционная семантика (ТОС)

Операционная семантика для тел вычислительных программ – это правила, которые позволяют формально проверить для каждой пары состояний и для каждого тела, является ли эта пара состояний входом-выходом данного тела. Если α – тело программы, а s' и s'' – пара состояний, которые входом-выходом α , то принято писать « $s' \langle \alpha \rangle s''$ ». Традиционно вход-выход $\langle \rangle$ определяется на состояниях рекурсивно по грамматической структуре тела программы следующим образом.

Определение традиционной операционной вычислительной семантики НеМо:

1. Присваивания и тесты:

- 1.1) для любых состояний s' и s'' и оператора присваивания « $x := t$ » имеем: $s' \langle x := t \rangle s'' \Leftrightarrow$ состояние s'' есть $UPD(s', x, s'(t))$ (т.е. в соответствии с неформальной семантикой выходное состояние s'' отличается от входного состояния s' только значением переменной x , которое теперь равно значению $s'(t)$ выражения t во входном состоянии s');
- 1.2) для любых состояний s' и s'' и теста « $\phi?$ » имеем: $s' \langle \phi? \rangle s'' \Leftrightarrow$ состояния s' и s'' совпадают и в этом состоянии выполнено свойство ϕ

(т.е. в соответствии с неформальной семантикой входное и выходное состояния совпадают в том и только том случае, когда условие теста верно, а в случае, когда условие не верно, входному состоянию не соответствует никакое выходное состояние).

2. Последовательная композиция и недетерминированный выбор:
 - 2.1) для любых состояний s' и s'' и любых тел α и β имеем: $s' \langle (\alpha ; \beta) \rangle s'' \Leftrightarrow$ существует такое состояние s , что $s' \langle \alpha \rangle s$ и $s \langle \beta \rangle s''$ (т.е. в соответствии с неформальной семантикой существует некоторое промежуточное состояние такое, что входное состояние и это промежуточное состояние являются входом-выходом α , и это промежуточное состояние и выходное состояние являются входом-выходом β);
 - 2.2) для любых состояний s' и s'' и любых тел α и β имеем: $s' \langle (\alpha \cup \beta) \rangle s'' \Leftrightarrow s' \langle \alpha \rangle s''$ или $s' \langle \beta \rangle s''$ (т.е. в соответствии с неформальной семантикой входное и выходное состояния или являются входом-выходом α , или являются входом-выходом β).
3. Недетерминированный цикл: для любых состояний s' и s'' и любого тела α имеем: $s' \langle (\alpha)^* \rangle s'' \Leftrightarrow$ существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \alpha \rangle s_{i+1}$ для любого $0 \leq i < n$ (т.е. в соответствии с неформальной семантикой существует такая конечная последовательность состояний, которая начинается с входного состояния, заканчивается выходным состоянием, а каждое из состояний в этой последовательности связано с непосредственно следующим за ним состоянием отношением вход-выход для тела α).

Структурная операционная семантика (COC)

При определении традиционной операционной семантики была сказана примечательная фраза, что вход-выход определяется на состояниях рекурсивно по грамматической структуре тела программ. Для языка HeMo это утверждение можно значительно усилить и добавить, что определение вход-выход для «подтел» не зависят от контекста (то есть ни друг от друга, ни от того, где они используются). Вообще говоря, структурная операционная семантика – это операционная семантика, определённая рекурсивно по грамматической структуре контекстно-независимым способом. В этом смысле определённая выше традиционная операционная семантика является структурной. Но существует традиция, сложившейся после основополагающих работ британского учёного Гордона Плоткина, представления структурной операционной семантики в виде аксиоматической системы.

Такой вариант структурной операционной семантики для тел программ на языке НеМо представлен ниже.

Определение структурной операционной вычислительной семантики НеМо:

$$\frac{}{s \langle x := t \rangle \text{UPD}(s, x, s(t))} ; \frac{}{s \langle \varphi ? \rangle s} , \text{ если } \varphi \text{ верно в состоянии } s ;$$

$$\frac{s' \langle \alpha \rangle s, s \langle \beta \rangle s''}{s' \langle (\alpha ; \beta) \rangle s''} ; \frac{s' \langle \alpha \rangle s''}{s' \langle (\alpha \cup \beta) \rangle s''} ; \frac{s' \langle \beta \rangle s''}{s' \langle (\alpha \cup \beta) \rangle s''}$$

$$\frac{}{s \langle (\alpha)^* \rangle s} ; \frac{s' \langle \alpha \rangle s, s \langle (\alpha)^* \rangle s''}{s' \langle (\alpha)^* \rangle s''}$$

В подходе Г. Плоткина объявляется, что пара состояний s' и s'' являются входом-выходом тела α тогда и только тогда, когда $s' \langle \alpha \rangle s''$ доказуемо в приведённой аксиоматической системе. Таким образом, для каждого тела и каждой пары состояний определены правила, которые позволяют проверить, является ли эта пара состояний входом-выходом программы (для этого достаточно осуществлять обратный поиск доказательства), причём, каждое правило сводит проблему контекстно-независимым способом к структурным фрагментам тела. Значит, семантика в стиле Г. Плоткина действительно является структурной и операционной.

Таким образом, мы имеем две формализации операционной семантики тела программ на языке НеМо: традиционную и структурную (по Г. Плоткину). Первая из них довольно-таки прозрачна и опирается на неформальное определение, а вторая носит более абстрактный характер и не связана явно с неформальной семантикой. Поэтому возникает естественный вопрос о связи этих семантик между собой и их отношении к неформальной семантике. Отвечает на этот вопрос следующее утверждение о взаимной полноте и непротиворечивости этих двух формальных операционных семантик.

Утверждение 2

Для любых состояний s' и s'' , для любого тела α программы на языке НеМо имеем: $s' \langle \alpha \rangle s''$ в традиционной операционной семантике тогда и толь-

ко тогда когда $s'\langle\alpha\rangle s''$ в структурной операционной семантике по Г. Плоткину.

Доказательство. Для того, что бы различать в доказательстве $s'\langle\alpha\rangle s''$ в ТОС и в СОС будем временно писать $s'\langle\alpha\rangle^T s''$ и $s'\langle\alpha\rangle^S s''$ соответственно. Сначала покажем, что $s'\langle\alpha\rangle^T s''$ влечёт $s'\langle\alpha\rangle^S s''$, а потом – что $s'\langle\alpha\rangle^S s''$ влечёт $s'\langle\alpha\rangle^T s''$.

$s'\langle\alpha\rangle^T s'' \Rightarrow s'\langle\alpha\rangle^S s''$ докажем индукцией по структуре тела программы.

1. База индукции состоит из двух случаев: присваивание и тест.
 - 1.1. $s'\langle x:=t \rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s'' = \text{UPD}(s', x, s'(t)) \Rightarrow$ (аксиома СОС для присваивания) $\Rightarrow s'\langle x:=t \rangle^S \text{UPD}(s', x, s'(t))$ и $s'' = \text{UPD}(s', x, s'(t)) \Rightarrow s'\langle x:=t \rangle^S s''$;
 - 1.2. $s'\langle \phi? \rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s' = s''$ и ϕ выполнено в этом состоянии \Rightarrow (аксиома СОС для теста) $\Rightarrow s'\langle \phi? \rangle^S s'$ и $s' = s'' \Rightarrow s'\langle \phi? \rangle^S s''$.

Индукционная гипотеза: предположим, что $s'\langle\alpha\rangle^T s'' \Rightarrow s'\langle\alpha\rangle^S s''$ верно для всех пар состояний и тел программ, которые являются составными частями α . Шаг индукции состоит из трёх подслучаев: для последовательной композиции, недетерминированного выбора и недетерминированной итерации.

2.

- 2.1. Пусть $\alpha \equiv (\beta ; \gamma)$. Имеем: $s'\langle\alpha\rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s'\langle\beta\rangle^T s$ и $s'\langle\gamma\rangle^T s''$ для некоторого состояния $s \Rightarrow$ (предположение индукции для β и γ) $\Rightarrow s'\langle\beta\rangle^S s$ и $s'\langle\gamma\rangle^S s''$ для некоторого состояния $s \Rightarrow$ (правило вывода СОС для последовательной композиции) $\Rightarrow s'\langle\alpha\rangle^S s''$.
- 2.2. Пусть $\alpha \equiv (\beta \cup \gamma)$. Имеем: $s'\langle\alpha\rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s'\langle\beta\rangle^T s''$ или $s'\langle\gamma\rangle^T s'' \Rightarrow$ (предположение индукции для β и γ) $\Rightarrow s'\langle\beta\rangle^S s''$ или $s'\langle\gamma\rangle^S s'' \Rightarrow$ (правило вывода СОС для недетерминированного выбора) $\Rightarrow s'\langle\alpha\rangle^S s''$.
3. Пусть $\alpha \equiv (\beta)^*$. Имеем: $s'\langle\alpha\rangle^T s'' \Rightarrow$ (определение ТОС) \Rightarrow существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \beta \rangle^T s_{i+1}$ для любого $0 \leq i < n$
 \Rightarrow (предположение индукции для β) \Rightarrow существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \beta \rangle^S s_{i+1}$ для любого $0 \leq i < n$
 \Rightarrow (аксиома СОС для недетерминированной итерации) \Rightarrow

существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \beta \rangle^{s_{i+1}}$ для любого $0 \leq i < n$ и, кроме того, $s_n \langle (\beta)^* \rangle^{s_n}$
 \Rightarrow (правило вывода СОС для недетерминированной итерации) \Rightarrow
существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \beta \rangle^{s_{i+1}}$ для любого $0 \leq i < (n-1)$ и, кроме того, $s_{n-1} \langle (\beta)^* \rangle^{s_n}$
 \Rightarrow (правило вывода СОС для недетерминированной итерации) \Rightarrow
 \dots
 \Rightarrow (правило вывода СОС для недетерминированной итерации) \Rightarrow
существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$ и $s_i \langle \beta \rangle^{s_{i+1}}$ для любого $0 \leq i < n$ и, кроме того, $s_0 \langle (\beta)^* \rangle^{s_n}$
 $\Rightarrow s' \langle \alpha \rangle^{s''}$.

$s' \langle \alpha \rangle^{s''} \Rightarrow s' \langle \alpha \rangle^T s''$ докажем индукцией по высоте дерева доказательства факта $s' \langle \alpha \rangle^{s''}$.

1. База индукции состоит из трёх случаев, когда высота дерева доказательства 0, соответствующих трём имеющимся аксиомам СОС: для присваивания, для теста и для недетерминированного выбора.
 - 1.1. $s \langle x := t \rangle^s \text{UPD}(s, x, s(t)) \Rightarrow$ (определение ТОС) \Rightarrow
 $s \langle x := t \rangle^T \text{UPD}(s, x, s(t));$
 - 1.2. $s \langle \phi ? \rangle^s \Rightarrow$ (аксиома СОС для теста) $\Rightarrow \phi$ выполнено в состоянии s
 \Rightarrow (определение ТОС) $\Rightarrow s \langle \phi ? \rangle^T s;$
 - 1.3. $s \langle (\alpha)^* \rangle^s \Rightarrow$ (определение ТОС для 0-кратного числа итераций) $\Rightarrow s$
 $\langle (\alpha)^* \rangle^T s.$

Индукционная гипотеза: предположим, что $s' \langle \alpha \rangle^{s''} \Rightarrow s' \langle \alpha \rangle^T s''$ верно для всех пар состояний и тел программ, для которых существует дерево доказательства $s' \langle \alpha \rangle^{s''}$ высоты не более $n \geq 0$. Шаг индукции состоит из трёх подслучаев, соответствующих правилам вывода: одного правила для последовательной композиции, двух аналогичных правил для недетерминированного выбора и одного правила для недетерминированной итерации.

2.
 - 2.1. Пусть $\alpha \equiv (\beta ; \gamma)$. Имеем: $s' \langle \alpha \rangle^{s''} \Rightarrow$ (правило вывода СОС для последовательной композиции) $\Rightarrow s' \langle \beta \rangle^s s$ и $s \langle \gamma \rangle^{s''} \Rightarrow$ (предположение индукции) $\Rightarrow s' \langle \beta \rangle^T s$ и $s \langle \gamma \rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s' \langle \alpha \rangle^T s''$.
 - 2.2. Пусть $\alpha \equiv (\beta \cup \gamma)$. Имеем: $s' \langle \alpha \rangle^{s''} \Rightarrow$ (одно из правил вывода СОС для недетерминированного выбора) $\Rightarrow s' \langle \beta \rangle^{s''}$ или $s' \langle \gamma \rangle^{s''} \Rightarrow$ (предположение индукции) $\Rightarrow s' \langle \beta \rangle^T s''$ или $s' \langle \gamma \rangle^T s'' \Rightarrow$ (определение ТОС) $\Rightarrow s' \langle \alpha \rangle^T s''$.

2.3. Пусть $\alpha \equiv (\beta)^*$. Имеем: $s' \langle \alpha \rangle^s s'' \Rightarrow$ (правила вывода СОС для недетерминированной итерации) $\Rightarrow s' = s''$ или $s' \langle \beta \rangle^s s$ и $s \langle (\beta)^* \rangle^s s''$ для некоторого состояния $s \Rightarrow$ (предположение индукции) $\Rightarrow s' \langle \beta \rangle^T s''$ или $s' \langle \beta \rangle^T s$ и $s \langle (\beta)^* \rangle^T s''$ для некоторого состояния $s \Rightarrow$ (определение ТОС) $\Rightarrow s' \langle \alpha \rangle^T s''$. ■

Приведённое доказательство объясняет, зачем нужна структурная операционная семантика в стиле Г. Плоткина: она позволяет устанавливать семантические свойства индукцией по высоте дерева семантического доказательства, что зачастую проще в математическом отношении, чем индукция по синтаксической структуре программы. Сравним, например, трудоёмкость (для человека) доказательств $s' \langle \alpha \rangle^T s'' \Rightarrow s' \langle \alpha \rangle^s s''$ и $s' \langle \alpha \rangle^s s'' \Rightarrow s' \langle \alpha \rangle^T s''$:

- в первом случае доказательство выполнено индукцией по синтаксической структуре, оно длиннее и использует (неявно) дополнительную индукцию по числу итераций в недетерминированной итерации;
- во втором случае доказательство выполнено индукцией по высоте дерева семантического доказательства, оно короче и не использует никакой скрытой дополнительной индукции.
- Сказанное позволяет заключить, что СОС в стиле Г. Плоткина – это, скорее математическая (т.е. ориентированная на человека), чем операционная (т.е. ориентированная на неодушевлённый компьютер) семантика.

Денотационная семантика (ДС) тела вычислительной программы

Ещё один вид математической семантики тел программ – это так называемая денотационная семантика (ДС).

Определение 7

Денотационная семантика – это алгебраическая семантика, которая сопоставляет каждому синтаксическому объекту элемент некоторой алгебры, а каждому конструктору новых синтаксических объектов – операцию этой же алгебры; и элемент алгебры, сопоставленный синтаксическому объекту, и операция, сопоставленная конструктору, называются в таком случае его денотацией.

Ниже мы познакомимся с денотационной семантикой тел вычислительных программ на языке НеМо. В этом случае «синтаксические объекты» – это тела программ, а «конструкторы» – это последовательная композиция

«;», недетерминированный выбор « \cup » и недетерминированная итерация « $*$ ».

Определение. Алгебра, в которой мы определим денотационную семантику НеМо-программ – это следующая алгебра бинарных отношений на пространстве состояний $(2^{SP \times SP}, \cup, \circ, *)$, где

- $2^{SP \times SP}$ – множество всех бинарных отношений на SP,
- \cup – операция объединения бинарных отношений,
- \circ – операция композиции бинарных отношений,
- $*$ – рефлексивное и транзитивное замыкание бинарных отношений.

(Краткое напоминание, что есть что в этой алгебре дано на следующих ниже «шпаргалках».)

Так вот, если α – тело программы, то бинарное отношение вход-выход как множество всех пар состояний, удовлетворяющих этому отношению, принято обозначать $[\alpha] : [\alpha] = \{ (s', s'') : s' \langle \alpha \rangle s'' \}$; примем это бинарное отношение в качестве денотации для α .

Утверждение 3

Для любых тел программ α и β имеют место следующие тождества в алгебре бинарных отношений на пространстве состояний $(2^{SP \times SP}, \cup, \circ, *)$:

1. $[(\alpha \cup \beta)] = [\alpha] \cup [\beta]$,
2. $[(\alpha ; \beta)] = [\alpha] \circ [\beta]$,
3. $[(\alpha)^*] = [\alpha]^*$.

Бинарное отношение - это множество пар состояний. Поэтому на бинарных отношениях определена обычная теоретико-множественная операция объединения:

$$P \cup Q = \{ (s', s'') : (s', s'') \in P \text{ и } (s', s'') \in Q \}.$$

Композиция бинарных отношений определяется в теоретико-множественных терминах следующим образом:

$$(P \circ Q) = \{ (u, w) : (u, v) \in P \text{ и } (v, w) \in Q \text{ для некоторого } v \}.$$

Рефлексивно-транзитивное замыкание бинарного отношения определяется в теоретико-множественных терминах таким образом:

$$P^* = (=) \cup (P) \cup (P \circ P) \cup (P \circ P \circ P) \cup \dots = \bigcup_{i \geq 0} P^i,$$

где P^0 - это тождественное бинарное отношение, P^0 - это само P , а P^{i+1} это $P \circ P^i$ для любого $i \geq 0$.

Примеры.

- «меньше» \cup «равно» = «небольшее»,
- «меньше» \cup «больше» = «неравно».

Пусть

- B - список пар населённых пунктов, между которыми есть автобусное сообщение

- A - список пар населённых пунктов, между которыми есть авиа сообщение

Тогда

- $B \circ A$ = список таких пар населённых пунктов, что из первого можно попасть в некоторый промежуточный пункт на автобусе, а из промежуточного - во второй - на самолёте;

- $A \circ B$ = список таких пар населённых пунктов, что из первого можно попасть в некоторый промежуточный пункт на самолёте, а из промежуточного - во второй - на автобусе.

Кроме того, имеем:

- A^0 = тождественное отношение = список пар населённых пунктов откуда куда можно «долететь» садясь в самолёт 0 раз (т.е. не садясь в самолёт вообще);
 - A^1 = список пар населённых пунктов откуда куда можно долететь садясь в самолёт 1 раз;
 - A^2 = список пар населённых пунктов откуда куда можно долететь садясь в самолёт 2 раз (с одной пересадкой);
 - A^3 = список пар населённых пунктов откуда куда можно долететь садясь в самолёт 3 раз (с двумя пересадками);
 - и т.д.
- Следовательно, $A^* =$ список пар населённых пунктов откуда куда можно долететь садясь в самолёт некоторое количество раз $n \geq 0$ раз (с $(n-1)$ пересадкой).

Доказательство тождеств выполним в порядке 1, 2, 3.

1. $[(\alpha \cup \beta)] =$ (по определению денотации тела программы)
 $= \{ (s', s'') : s' \langle (\alpha \cup \beta) \rangle s'' \} =$ (по определению ТОС)
 $= \{ (s', s'') : s' \langle \alpha \rangle s'' \text{ или } s' \langle \beta \rangle s'' \} = \{ (s', s'') : s' \langle \alpha \rangle s'' \} \cup \{ (s', s'') : s' \langle \beta \rangle s'' \} =$
 $(\text{по определению денотации тела программы})$
 $= [\alpha] \cup [\beta];$
2. $[(\alpha ; \beta)] =$ (по определению денотации тела программы)
 $= \{ (s', s'') : s' \langle (\alpha ; \beta) \rangle s'' \} =$ (по определению ТОС)
 $= \{ (s', s'') : s' \langle \alpha \rangle s \text{ и } s \langle \beta \rangle s'' \text{ для некоторого } s \} =$
 $(\text{по определению денотации тела программы})$
 $= \{ (s', s'') : (s, s') \in [\alpha] \text{ и } (s, s'') \in [\beta] \text{ для некоторого } s \} =$
 $(\text{по определению композиции бинарных отношений})$
 $= [\alpha] \circ [\beta];$
3. $[(\alpha)^*] =$ (по определению денотации тела программы)
 $= \{ (s', s'') : s' \langle (\alpha)^* \rangle s'' \} =$ (по определению ТОС)
 $= \{ (s', s'') : \text{существует такое } n \geq 0 \text{ и последовательность состояний } s_0, \dots, s_n, \text{ что } s_0 = s', s_n = s'' \text{ и } s_i \langle \alpha \rangle s_{i+1} \text{ для любого } 0 \leq i < n \} =$

$$\begin{aligned}
&= \bigcup_{n \geq 0} \{ (s', s'') : \text{существует последовательность состояний } s_0, \dots, s_n, \\
&\text{что } s_0=s', s_n=s'' \text{ и } s_i \langle \alpha \rangle s_{i+1} \text{ для любого } 0 \leq i < n \} = \\
&\quad (\text{по определению денотации тела программы}) \\
&= \bigcup_{n \geq 0} \{ (s', s'') : \text{существует последовательность состояний } s_0, \dots, s_n, \\
&\text{что } s_0=s', s_n=s'' \text{ и } (s_i, s_{i+1}) \in [\alpha] \text{ для любого } 0 \leq i < n \} = \\
&\quad (\text{по определению степени бинарного отношения}) \\
&= \bigcup_{n \geq 0} \{ (s', s'') : (s', s'') \in [\alpha]^n \} = \bigcup_{n \geq 0} [\alpha]^n = \\
&\quad (\text{по определению рефлексивного-транзитивного замыкания}) \\
&= [\alpha]^*. \blacksquare
\end{aligned}$$

Утверждение 2 позволяет определить денотационную семантику тел программ на языке HeMo в чисто алгебраической манере, а именно: определить денотации для «элементарных» тел (т.е. для присваиваний и тестов) и для конструкторов (т.е. для «;», « \cup » и « $*$ »), а для «составных» тел (т.е. построенных из присваиваний и тестов при помощи конструкторов) – «вычислять» на основе тождеств из утверждения 2; таким образом определённую денотационную семантику тел программ и конструкторов тел программ будем по-прежнему обозначать посредством « $[\]$ ».

Определение денотационной вычислительной семантики HeMo:

1.
 - 1.1. $[x:=t] = \{ (s, \text{UPD}(s, x, s(t))) : s \in \text{SP} \}$;
 - 1.2. $[\phi ?] = \{ (s, s) : \text{в } s \text{ верно } \phi \}$;
2.
 - 2.1. $[\cup] =$ операция объединения отношений « \cup »
(т.е. операция $\lambda P, Q \subseteq \text{SP} \times \text{SP}. (P \cup Q)$);
 - 2.2. $[;] =$ операция композиции отношений « $^\circ$ »
(т.е. операция $\lambda P, Q \subseteq \text{SP} \times \text{SP}. (P^\circ Q)$);
 - 2.3. $[*] =$ операция рефлексивно-транзитивного замыкания отношений
« $*$ »
(т.е. операция $\lambda P \subseteq \text{SP} \times \text{SP}. (P^*)$);
3.
 - 3.1. $[(\alpha \cup \beta)] = [\cup] ([\alpha], [\beta]) = [\alpha] \cup [\beta]$,
 - 3.2. $[(\alpha ; \beta)] = [^\circ] ([\alpha], [\beta]) = [\alpha]^\circ [\beta]$,
 - 3.3. $[(\alpha)^*] = [*]([\alpha]) = [\alpha]^*$.

В рамках денотационного подхода к семантике тел вычислительных программ объявляется, что состояния s' и s'' являются входом-выходом тела

α тогда и только тогда, когда $(s', s'') \in [\alpha]$; таким образом мы имеем ещё одну формализацию отношения $s' \langle \alpha \rangle s''$. В утверждении 1 мы уже выяснили, что две первых формализации (ТОС и СОС) совпадают. Поэтому возникает естественный вопрос о связи денотационной семантики с эти двумя операционными. Отвечает на этот вопрос следующее утверждение о полноте и непротиворечивости денотационной семантики тел программ на языке НеМо.

Утверждение 4

Для любых состояний s' и s'' , для любого тела α программы на языке НеМо имеем: $s' \langle \alpha \rangle s''$ в традиционной/структурной операционной семантике тогда и только тогда когда $s' \langle \alpha \rangle s''$ в денотационной семантике.

Доказательство этого утверждения следует из утверждения 2. ■

Зачем нужна денотационная семантика

Утверждение 3 может привести к ошибочному выводу, что денотационная семантика тела вычислительных программ не добавила ничего нового к нашим знаниям о семантике тела по сравнению с операционной семантикой. Это отнюдь не так, ибо «за кадром» утверждения 3 остались денотации для конструкторов, о которых это утверждение вообще ничего не говорит, т.к. денотации конструкторов просто не с чем сравнивать в операционной семантике. Зато в денотационной семантике тел вычислительных программ теперь появляется возможность определить понятие «управления» (или «потока управления» – «control flow») следующим образом.

Определение потока управления в вычислительной программе НеМО. Пусть α – тело вычислительной программы. Предположим, что у нас есть некоторое «отношение семантической неразличимости» на операторах (присваиваниях и тестах), которые встречаются в α . (Это «отношение семантической неразличимости» может быть, например, синтаксическое совпадение, или совпадение левых частей и равенство правых и т.п.) Перенумеруем все операторы в α с точностью до этого «отношения семантической неразличимости»: сопоставим неразличимым операторам один и тот же номер, а различимым – разные номера. Пусть классы неразличимых операторов получили номера от 1 до некоторого $n \geq 1$, а σ_i и τ_i обозначают денотацию и переменную для денотации операторов из класса, получившего номер $i \in [1..n]$. Заменим каждый оператор в α на соответствующую ему переменную τ_i , а каждый конструктор – на его денотацию; получившееся выражение алгебры бинарных отношений на SP с переменными τ_1, \dots, τ_n называется потоком управления тела α и обозначается $cf(\alpha)$: это выражение опре-

деляет какие из операторов (с точностью до семантической эквивалентности операторов) когда и в каком порядке выполняются в α .

Имеем очевидное тождество $[\alpha] = ((\lambda\tau_1, \dots, \tau_n \cdot \text{cf}(\alpha)) \sigma_1, \dots, \sigma_n)$, означающее, что если в α заменить все операторы и конструкторы на их семантику (денотации) и вычислить получившееся выражение алгебры бинарных отношений, то получится семантика (денотации) α .

Таким образом, поток управления для тела программы α - это некоторое такое выражение $\text{cf}(\alpha)$ алгебры бинарных отношений с переменными $\underline{\tau}$ для семантики операторов $\underline{\sigma}$, что имеет место равенство $[\alpha] = ((\lambda\underline{\tau}. \text{cf}(\alpha)) \underline{\sigma})$. Но для выражений алгебры бинарных отношений с переменными существуют тождественные преобразования, которые верны при любых значениях входящих переменных. Следовательно, если применить любую цепочку таких преобразований к $\text{cf}(\alpha)$, то получится некоторое выражение ε этой же алгебры, которое будет эквивалентно $\text{cf}(\alpha)$, и, следовательно, будет удовлетворять следующему равенству $[\alpha] = ((\lambda\underline{\tau}. \varepsilon) \underline{\sigma})$.

Поэтому появляется возможность изучать эквивалентные преобразования потоков управления тел программ с целью, например, большей наглядности для человека или большей эффективности при реализации на компьютере. Следующее утверждение содержит несколько наиболее важных тождеств для потоков управления тел вычислительных программ языка HeMo.

Утверждение 5

В алгебре бинарных отношений на пространстве состояний $(2^{\text{SP} \times \text{SP}}, \cup, \circ, *)$ выполняются следующие тождества для выражений с переменными τ_1, τ_2 и τ_3 для бинарных отношений:

1. коммутативность объединения $\tau_1 \cup \tau_2 = \tau_2 \cup \tau_1$,
2. ассоциативность объединения $(\tau_1 \cup \tau_2) \cup \tau_3 = \tau_1 \cup (\tau_2 \cup \tau_3)$,
3. ассоциативность композиции $(\tau_1 \circ \tau_2) \circ \tau_3 = \tau_1 \circ (\tau_2 \circ \tau_3)$,
4. левая дистрибутивность композиции относительно объединения

$$\tau_1 \circ (\tau_2 \cup \tau_3) = (\tau_1 \circ \tau_2) \cup (\tau_1 \circ \tau_3),$$
5. правая дистрибутивность композиции относительно объединения

$$(\tau_1 \cup \tau_2) \circ \tau_3 = (\tau_1 \circ \tau_3) \cup (\tau_2 \circ \tau_3),$$
6. раскрутка рефлексивного и транзитивного замыкания $\tau_1^* = (=) \cup \tau_1 \circ (\tau_1^*)$, где $(=)$ обозначает тождественное отношение.

Доказательство этого утверждения получается расписыванием в теоретико-множественных терминах бинарных отношений как множеств пар состояний. ■

Для нас очень полезными для понимания тел программ человеком являются тождества 2 и 3 из утверждения 4, которые говорят, что порядок выполнения среди нескольких подряд идущих объединений или композиций не важен: с точки зрения потока управления это всё эквивалентно и приводит к телам с совпадающей семантикой вход-выход. Поэтому мы будем при обсуждении тел программ для удобочитаемости опускать многие скобки, во круг подряд идущих конструкторов последовательной композиции «;» и недетерминированного выбора « \cup », семантика которых как раз композиция и объединение отношений.

Пример.

Пусть у нас есть три целочисленных переменных x , y и z . Тогда состояния естественно представлять тройками целых чисел следующим образом: тройка (a, b, c) представляет состояние s , в котором $s(x) = a$, $s(y) = b$ и $s(z) = c$. Пусть α - это следующее тело программы:

$$((x := 0 ; y := 0) ; (((y \leq z ? ; (y := y + x ; y := y + x)) ; (y := y + 1 ; x := x + 1))))^* .$$

Только два оператора являются семантически неразличимыми в этом теле: это два разных оператора « $y := y + x$ ». Поэтому примем следующую нумерацию семантически различных операторов, вычислим их денотации и введём соответствующие переменные:

номер	оператор	Денотация	Переменная
1	$x := 0$	$\sigma_1 = \{((a, b, c), (0, b, c)) : a, b, c \in \text{INT}\}$	τ_1
2	$y := 0$	$\sigma_2 = \{((a, b, c), (a, 0, c)) : a, b, c \in \text{INT}\}$	τ_2
3	$y \leq z ?$	$\sigma_3 = \{((a, b, c), (a, b, c)) : b \leq c\}$	τ_3
4	$y := y + x$	$\sigma_4 = \{((a, b, c), (a, b + a, c)) : a, b, c \in \text{INT}\}$	τ_4
5	$y := y + 1$	$\sigma_5 = \{((a, b, c), (a, b + 1, c)) : a, b, c \in \text{INT}\}$	τ_5
6	$x := x + 1$	$\sigma_6 = \{((a, b, c), (a + 1, b, c)) : a, b, c \in \text{INT}\}$	τ_6

Следовательно, поток управления тела α есть

$$cf(\alpha) = ((\tau_3 \circ \tau_2) \circ (((\tau_3 \circ (\tau_4 \circ \tau_4)) \circ (\tau_5 \circ \tau_6))))^*).$$

В силу тождеств 2 и 3 из утверждения 4 большинство скобок в этом выражении можно опустить, что не изменит значения этого выражения, но повысит удобочитаемость для человека:

$$cf(\alpha) = \tau_3 \circ \tau_2 \circ (\tau_3 \circ \tau_4 \circ \tau_4 \circ \tau_5 \circ \tau_6)^*.$$

Поэтому исходное тело вычислительной программы для человека тоже удобнее представлять в следующем виде

$$x:= 0 ; y:= 0 ; (y \leq z ? ; y:= y+x ; y:= y+x ; y:= y+1 ; x:= x+1)^*$$

а не в исходном виде

$$((x:= 0 ; y:= 0) ; (((y \leq z ? ; (y:= y+x ; y:= y+x)) ; (y:= y+1 ; x:= x+1))))^*).$$

Формальная семантика вычислительных НеМо-программ

Теперь всё готово для определения формальной семантики вычислительных программ на языке НеМо.

Определение семантики вычислительной НеМо-программы. Пусть π - произвольная вычислительная НеМо-программа. Она (как и всякая программа на языке НеМо) состоит из описаний и тела: $\pi \equiv \delta \beta$, где δ - описания, а β - тело. По совокупности описаний δ можно определить пространство состояний, которое выше при фиксированном δ обозначалось SP , а в случае, когда (для определённости) необходимо указать δ явно, будет обозначаться $SP(\delta)$. Определим для программы π пространство состояний $SP(\pi)$ как пространство состояний её совокупности описаний $SP(\delta)$. В терминах этого пространства состояний выше было формально определено бинарное отношение вход-выход $\langle \beta \rangle$ на $SP(\pi) \times SP(\pi)$; для этого были развиты три формализма (ТОС, СОС и ДС), которые оказались полностью согласованными между собой (утверждения 1 и 3 о взаимной полноте и непротиворечивости) и с неформальной семантикой языка. Поэтому определим для вычислительной программы π бинарное отношение вход-выход $\langle \pi \rangle$ на $SP(\pi) \times SP(\pi)$ как бинарное отношение $\langle \beta \rangle$ на $SP(\pi) \times SP(\pi)$. Под операционной семантикой вычислительной программы π мы будем понимать пару, состоящую из пространства состояний $SP(\pi)$ и из бинарного отношения $\langle \pi \rangle$ на $SP(\pi) \times SP(\pi)$.

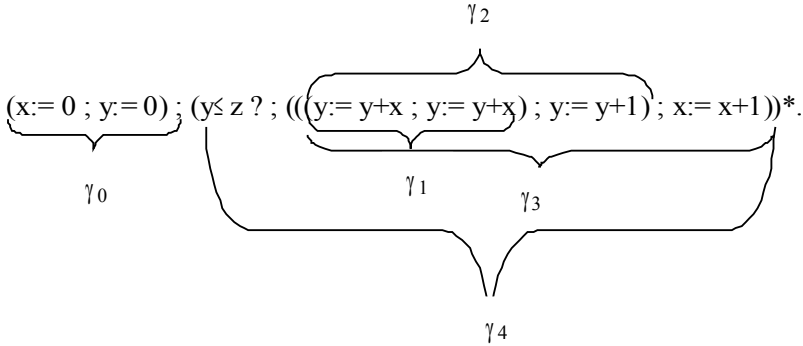
Пример. Построим семантику вычислительной программы, которая уже немного обсуждалась выше:

VAR $x : \text{INT}$; VAR $y : \text{INT}$; VAR $z : \text{INT}$;
 $((x := 0 ; y := 0) ; (((y \leq z ? ; (y := y+x ; y := y+x)) ; (y := y+1 ; x := x+1))))^*$.

Обозначим её саму посредством π , её описания – посредством δ , а тело – посредством β . По определению $\text{SP}(\pi) = \text{SP}(\delta)$, поэтому состояния этой программы π естественно представлять тройками целых чисел следующим образом как это было сделано выше. Опять же по определению $\langle \pi \rangle = \langle \beta \rangle$ на пространстве состояний $\text{SP}(\pi)$. Однако, нам удобнее построить не семантику (денотацию) тела β , а эквивалентного ему (в силу эквивалентного преобразования потока управления, а именно – ассоциативности последовательной композиции) следующего тела γ :

$(x := 0 ; y := 0) ; (y \leq z ? ; (((y := y+x ; y := y+x) ; y := y+1) ; x := x+1))^*$.

Для удобства введём обозначения γ_0 , γ_1 , γ_2 , γ_3 и γ_4 для фрагментов тела γ так как это сделано на следующем рисунке:



Имеем:

1. $[\gamma_0] = \sigma_1 \circ \sigma_2 =$
 $= \{ ((a,b,c) , (0,b,c)) : a,b,c \in \text{INT} \} \circ \{ ((a,b,c) , (a,0,c)) : a,b,c \in \text{INT} \}$
 $=$
 $= \{ ((a,b,c) , (0,0,c)) : a,b,c \in \text{INT} \};$

2. $[\gamma_1] = \sigma_4 \circ \sigma_4 =$
 $= \{ ((a,b,c), (a,b+a,c)) : a,b,c \in \text{INT} \} \circ \{ ((a,b,c), (a,b+a,c)) : a,b,c \in \text{INT} \} =$
 $= \{ ((a,b,c), (a, (b+2a), c)) : a,b,c \in \text{INT} \};$
3. $[\gamma_2] = [\gamma_1] \circ \sigma_5 =$
 $= \{ ((a,b,c), (a, (b+2a), c)) : a,b,c \in \text{INT} \} \circ \{ ((a,b,c), (a,b+1,c)) : a,b,c \in \text{INT} \} =$
 $= \{ ((a,b,c), (a, (b+2a+1), c)) : a,b,c \in \text{INT} \};$
4. $[\gamma_3] = [\gamma_2] \circ \sigma_6 =$
 $= \{ ((a,b,c), (a, (b+2a+1), c)) : a,b,c \in \text{INT} \} \circ \{ ((a,b,c), (a+1,b,c)) : a,b,c \in \text{INT} \} =$
 $= \{ ((a,b,c), ((a+1), (b+2a+1), c)) : a,b,c \in \text{INT} \};$
5. $[\gamma_4] = \sigma_3 \circ [\gamma_3] =$
 $= \{ ((a,b,c), (a,b,c)) : b \leq c \} \circ \{ ((a,b,c), ((a+1), (b+2a+1), c)) : a,b,c \in \text{INT} \} =$
 $= \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \};$
6. $[\gamma_4]^0 = \{ ((a,b,c), (a,b,c)) : a,b,c \in \text{INT} \},$
 $[\gamma_4]^1 = [\gamma_4] = \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \},$
 $[\gamma_4]^2 = [\gamma_4] \circ [\gamma_4]^1 =$
 $= \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \} \circ \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \} = \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \} \circ$
 $\{ (((a+1), (b+2a+1), c), ((a+2), (b+4a+4), c)) : b+2a+1 \leq c \} =$
 $= \{ ((a,b,c), ((a+2), (b+4a+3), c)) : b \leq c \text{ и } (b+2a+1) \leq c \} =$
 $= \{ ((a,b,c), ((a+2), (b+4a+3), c)) : b, (b+2a+1) \leq c \},$
 \dots
 $[\gamma_4]^{i+1} = [\gamma_4] \circ [\gamma_4]^i =$
 $= \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \} \circ$
 $\{ (((a+1), (b+2a+1), c), ((a+i), (b+2ia+i^2), c)) : b, \dots, (b+2(i-1)a + (i-1)^2) \leq c \} =$
 $= \{ ((a,b,c), ((a+1), (b+2a+1), c)) : b \leq c \} \circ$
 $\{ (((a+1), (b+2a+1), c), ((a+i+1), (b+2(i+1)a+(i+1)^2), c)) : b, \dots, (b+2ia + i^2) \leq c \} =$
 $= \{ ((a,b,c), ((a+(i+1)), (b+2(i+1)a+(i+1)^2), c)) : b, \dots, (b+2ia + i^2) \leq c \};$
7. $[\gamma_4]^* = \cup_{i \geq 0} [\gamma_4]^i = \{ ((a,b,c), (a,b,c)) : a,b,c \in \text{INT} \} \cup$
 $\cup_{i \geq 1} \{ ((a,b,c), ((a+i), (b+2ia+i^2), c)) : b, \dots, (b+2(i-1)a + (i-1)^2) \leq c \};$
8. $[\gamma] = [\gamma_0] \circ [\gamma_4]^* = \{ ((a,b,c), (0,0,c)) : a,b,c \in \text{INT} \} \circ$

$$\begin{aligned} & \{ ((a,b,c), (a,b,c)) : a,b,c \in \text{INT} \} \cup \\ & \cup_{i \geq 1} \{ ((a,b,c), ((a+i), (b+2ia+i^2), c)) : b, \dots, (b+2(i-1)a+(i-1)^2) \leq c \} = \\ & = \{ ((a,b,c), (0,0,c)) : a,b,c \in \text{INT} \} \cup (\cup_{i \geq 1} \{ ((a,b,c), (i, i^2, c)) : (i-1)^2 \leq c \}). \end{aligned}$$

Таким образом, семантика тела γ , ему эквивалентного тела β и, тем самым, отношение вход-выход для вычислительной программы π построены: получив входное состояние $(a,b,c) \in \text{INT}$ программа останавливается или в выходном состоянии $(0,0,c)$, или (при $c \geq 0$) в выходном состоянии (i, i^2, c) , где $0 \leq i$ и $(i-1)^2 \leq c$.

Лекция 14. Введение в компиляцию языков программирования (на примере языка HeMo)

Проблема трансляция для HeMo-программ

Неформально говоря, исполнение на реальной (т.н. объектной) машине программы на языке виртуальной HeMo-машины – это её интерпретации. На формальном уровне интерпретация – это процесс пошагового построения расширенного семантического дерева (например, методом ветвей и границ) при котором шаг – это срабатывание (оператора) программы. Процесс интерпретации для языка виртуальной HeMo-машины возможен потому, что тело любой программы виртуальной HeMo-машины – это неструктурированное множество операторов, и каждый оператор соответствует некоторой фиксированной и независимой от других операторов совокупности действий объектной вычислительной машины по преобразованию данных и по передаче управления.

Тело любой программы на языке HeMo тоже состоит из операторов (присваиваний и тестов), но уже не сводится просто к множеству операторов, а представляет некоторую структуру, построенную из операторов при помощи конструкторов последовательного исполнения «;», недетерминированного выбора « \cup » и недетерминированного цикла «*». В HeMo каждый оператор отвечает только за преобразования данных, а за передачу управления отвечает поток управления, который как-раз определяется структурой тела программы. Поэтому к программам на языке HeMo процесс интерпретации непосредственно не применим, и, значит, программы на языке HeMo не могут быть непосредственно исполнены ни на какой объектной машине. Следовательно, необходима перевод (трансляция) программ на языке

HeMo в эквивалентные программы на языке, к которому процесс интерпретации непосредственно применим.

Разумеется, мы будем транслировать программы на языке HeMo в эквивалентные программы на языке виртуальной HeMo-машины, для которых процесс интерпретации уже обсуждался нами ранее. Алгоритм трансляции будет описан ниже. Однако, т.к. этот алгоритм должен транслировать программы эквивалентным образом (т.е. с сохранением семантики программ), то после описания алгоритма нам придётся доказать формальное утверждение о корректности этого алгоритма. Поэтому нам придётся пока ограничиться только классом вычислительных программ, для которых уже была построена формальная семантика как в языке HeMo, так и в языке виртуальной HeMo-машины.

Трансляция языка HeMo в язык виртуальной HeMo-машины

Алгоритм трансляции получает на вход грамматически разобрannую программу на языке HeMo, которая состоит из описаний и тела, а выдаёт – программу на языке виртуальной HeMo-машины, которая состоит из множества описаний и множества операторов. Алгоритм использует рекурсивную процедуру-функцию BODY-TO-SET («тело в множество» или, для краткости, B2S). Алгоритм возвращает программу на языке виртуальной HeMo-машины, а процедура – множество операторов языка виртуальной HeMo-машины. Алгоритм и процедура будут использовать полуформальное понятие «синтаксически разобрannая» программа и «синтаксически разобрannое» тело, которые означают не только синтаксическую корректность, но и доступ к грамматической информации о составных частях программы и тела, которую предоставляет дерево грамматического разбора как-то: в программе выделены описания и тело, в теле – «подтела» и конструктор (если это последовательная композиция, недетерминированный выбор или итерация), изменяемая переменная и выражение правой части (если это оператор присваивания) или условие (если это тест).

Алгоритм трансляции

[Вход является синтаксически разобрannая вычислительная программа π на языке HeMo.] // Предусловие.

Пусть δ – множество всех описаний программы π , а β – её тело. Тогда пусть α – программа на языке виртуальной HeMo-машины, состоящая из множества описаний δ и множества операторов B2S(β).

[Выходом является вычислительная программа α на языке виртуальной HeMo-машины такая, что её вычислительная семантика $[\alpha]$ совпадает с де-

нотационной вычислительной семантикой $[\pi]$ для вычислительной программы π на языке HeMo.] // Постусловие.

Процедура BODY-TO-SET (β) использует три специальных операции «тах», «+» и «/»:

- если S - множество операторов языка виртуальной HeMo-машины, то тах S выбирает максимальную по числовому значению метку, которая встречается во множестве операторов S ;
- если S - множество операторов языка виртуальной HeMo-машины, а n – число (в частности, метка), то $(S + n)$ – это множество S , в котором каждое вхождение каждой метки l заменяется на $(l + n)$;
- если S - множество операторов языка виртуальной HeMo-машины, n – число (в частности, метка), а l – метка, то $S(n/l)$ – это множество S , в котором каждое вхождение данной метки l заменяется на данное число n .

[Входной параметр β является синтаксически разобранным телом – фрагментом тела вычислительной программы π на языке HeMo, поданной на вход алгоритма трансляции.] // Предусловие.

1. Если β - это некоторый оператор присваивания $x := t$,
то $B2S := \{(0: x := t \text{ goto } \{1\})\}$.
2. Если β - это некоторый тест ϕ ?, то $B2S := \{(0: \text{if } \phi \text{ then } \{1\} \text{ else } \emptyset)\}$.
3. Если β - это последовательная композиция $(\theta ; \xi)$, то выполнить следующее:
 - 3.1. $S_\theta := B2S(\theta)$;
 - 3.2. $\text{end}_\theta := \max S_\theta$;
 - 3.3. $S_\xi := B2S(\xi) + \text{end}_\theta$;
 - 3.4. $B2S := S_\theta \cup S_\xi$.
4. Если β - это недетерминированный выбор $(\theta \cup \xi)$, то выполнить следующее:
 - 4.1. $S_\theta := B2S(\theta) + 1$;
 - 4.2. $\text{end}_\theta := \max S_\theta$;
 - 4.3. $S_\xi := B2S(\xi) + \text{end}_\theta$;
 - 4.4. $\text{end}_\xi := \max S_\xi$;
 - 4.5. $B2S := \{(0: \text{if TRUE then } \{1, \text{end}_\theta\} \text{ else } \emptyset)\} \cup S_\theta(\text{end}_\xi/\text{end}_\theta) \cup S_\xi$.
5. Если β - это недетерминированная итерация $(\theta)^*$, то выполнить следующее:
 - 5.1. $S_\theta := B2S(\theta) + 1$;
 - 5.2. $\text{end}_\theta := \max S_\theta$;

5.3. $B2S := \{(0: \text{if TRUE then } \{1, \text{end}_\theta\} \text{ else } \emptyset)\} \cup S_\theta(0/\text{end}_\theta)$.

$[B2S(\beta)]$ является множеством операторов языка виртуальной НеМо-машины, обладающим следующими свойствами:

- метки, которые встречаются в $B2S(\beta)$, образуют интервал $[0.. \max B2S(\beta)]$, причём, $\max B2S(\beta)$ является единственной меткой, которая встречается в $B2S(\beta)$, но не метит ни одного оператора в $B2S(\beta)$;
 - переменные, которые встречаются в $B2S(\beta)$, описаны в описаниях δ программы π , причём, они используются в $B2S(\beta)$ в соответствии с их типами;
 - вычислительная семантика $[\alpha_\beta]$ программы α_β виртуальной НеМо-машины, состоящей из множества описаний δ и множества операторов $B2S(\beta)$, совпадает с денотационной вычислительной семантикой $[\beta]$ тела вычислительной программы β на пространстве состояний $SP(\delta)$
-] // Постусловие.

Для того, чтобы сделать процедуру BODY-TO-SET более наглядной и понятной, представим её графически:

$$B2S(x := t) = \begin{array}{c} \text{входная} \\ \text{метка } 0 \end{array} \rightarrow \boxed{x := t} \begin{array}{c} \text{выходная} \\ \text{метка } 1 \end{array}$$

$$B2S(\phi \text{ ?}) = \begin{array}{c} \text{входная} \\ \text{метка } 0 \end{array} \rightarrow \begin{array}{c} \text{ } \\ \phi \\ \text{ } \end{array} \begin{array}{c} \text{выходная} \\ \text{метка } 1 \end{array} +$$

$$B2S(\theta \text{ ; } \xi) = \begin{array}{c} \text{входная} \\ \text{метка } 0 \end{array} \rightarrow B2S(\theta) \xrightarrow[\text{end}_\theta]{\text{метка}} B2S(\xi) + \text{end}_\theta \xrightarrow[\text{end}_\theta + \text{end}_\xi]{\text{выходная метка}}$$

$$B2S(\theta \cup \xi) = \begin{array}{c} \text{метка } \text{end}_\theta \\ \text{входная} \\ \text{метка } 0 \end{array} \begin{array}{l} \nearrow B2S(\xi) + \text{end}_\theta \xrightarrow[\text{end}_\theta + \text{end}_\xi]{\text{метка}} \\ \searrow (B2S(\theta) + 1) \xrightarrow[\text{end}_\theta]{\text{метка}} \end{array} \begin{array}{c} \text{выходная метка} \\ \text{end}_\theta + \text{end}_\xi \\ (\text{end}_\theta + \text{end}_\xi) / \text{end}_\theta \end{array}$$

$$B2S(\theta *) = \begin{array}{c} \text{входная} \\ \text{метка } 0 \end{array} \begin{array}{l} \nearrow \text{выходная метка } \text{end}_\theta \\ \searrow (B2S(\theta) + 1) \xrightarrow[\text{end}_\theta]{\text{метка}} 0 / \text{end}_\theta \end{array}$$

Утверждение 6

Процедура BODY-TO-SET является тотально корректной относительно своих пред- и пост- условий.

Доказательство проведём индукцией по синтаксической структуре тела β .

1. База индукции: β - это или присваивание, или тест.

1.1. Пусть β - это некоторое присваивание $x := t$. Согласно определению, $B2S(\beta)$ использует только две метки 0 и 1, причём, метка 1 не метит никакого оператора. Пусть α_β - программа виртуальной НеМо-машины, которая состоит из описаний δ и множества операторов $B2S(\beta)$. В пространстве состояний $SP(\delta)$ имеем:

$$\begin{aligned} [\alpha_\beta] &= \\ &= \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\beta, \text{ начинающаяся в конфигурации } (0, s'), \text{ а заканчивающаяся в конфигурации } (1, s'') \} = \\ &= \{ (s', s'') : s'' = \text{UPD}(s', x, s'(t)) \} = \{ (s, \text{UPD}(s, x, s(t))) : s \in SP(\delta) \} = \\ &= [\beta]. \end{aligned}$$

1.2. Пусть β - это некоторый тест ϕ ?. Согласно определению, $B2S(\beta)$ использует только две метки 0 и 1, причём, метка 1 не метит никакого оператора. Пусть α_β - программа виртуальной НеМо-машины, которая состоит из описаний δ и множества операторов $B2S(\beta)$. В пространстве состояний $SP(\delta)$ имеем:

$$\begin{aligned} [\alpha_\beta] &= \\ &= \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\beta, \text{ начинающаяся в конфигурации } (0, s'), \text{ а заканчивающаяся в конфигурации } (1, s'') \} = \\ &= \{ (s, s) : s \models \phi \} = \{ (s, s) : s \text{ верно } \phi \} = \\ &= [\beta]. \end{aligned}$$

Индукционная гипотеза: предположим, что процедура $B2S$ тотально корректна для всех тел, которые являются фрагментами тела β .

2. Шаг индукции: β - это или последовательная композиция, или недетерминированный выбор, или недетерминированная итерация.

2.1. Пусть $\beta \equiv (\theta ; \xi)$. Т.к. θ является фрагментом β , то согласно предположению индукции, $B2S(\theta)$ использует метки в интервале $[0.. \max B2S(\theta)]$, причём, метка $\max B2S(\theta)$ является её единственной выходной меткой. Аналогично, т.к. ξ является фрагментом β , то согласно предположению индукции, $B2S(\xi)$ использует метки в интервале $[0.. \max B2S(\xi)]$, причём, метка $\max B2S(\xi)$ является её единственной выходной меткой; следовательно, $B2S(\xi) + \max B2S(\theta)$ использует метки в интервале $[\max B2S(\theta).. (\max B2S(\theta) + \max B2S(\xi))]$, причём, метка $(\max B2S(\theta) + \max B2S(\xi))$ является её единственной выходной меткой. Т.к. $B2S(\beta) = B2S(\theta) \cup (B2S(\xi) + \max B2S(\theta))$, то $B2S(\beta)$ использует метки в интервале $[0.. (\max B2S(\theta) + \max B2S(\xi))]$, причём, метка $(\max B2S(\theta) + \max B2S(\xi))$ является её единственной выходной меткой.

Пусть α_β - программа виртуальной НеМо-машины, которая состоит из описаний δ и множества операторов $B2S(\beta)$. В пространстве состояний $SP(\delta)$ имеем:

$$\begin{aligned}
 [\alpha_\beta] &= \\
 &= \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\beta, \text{ которая начинается в конфигурации } (0, s'), \text{ а заканчивается в конфигурации } ((\max B2S(\theta) + \max B2S(\xi)), s'') \} = \\
 &= \{ (s', s'') : \text{существует состояние } s \text{ и конечная полная трасса } \alpha_\beta, \text{ которая начинается в конфигурации } (0, s'), \text{ проходит через конфигурацию } (\max B2S(\theta), s), \text{ а заканчивается в конфигурации } ((\max B2S(\theta) + \max B2S(\xi)), s'') \} = \\
 &= \{ (s', s'') : \\
 &\quad \text{существует такие состояние } s \text{ и конечные полные трассы } \alpha_\theta \text{ и } \alpha_\xi, \\
 &\quad \text{что конечная полная трасса } \alpha_\theta \text{ начинается в } (0, s') \text{ и заканчивается в } (\max B2S(\theta), s), \text{ а конечная полная трасса } \alpha_\xi \text{ начинается в } (0, s) \text{ и заканчивается в } (\max B2S(\xi), s'') \} = \\
 &= \{ (s', s) : \text{существует конечная полная трасса } \alpha_\theta, \text{ которая начинается в } (0, s') \text{ и заканчивается в } (\max B2S(\theta), s) \} \circ \{ (s, s'') : \text{существует конечная полная трасса } \alpha_\xi, \text{ которая начинается в } (0, s) \text{ и заканчивается в } (\max B2S(\xi), s'') \} = \\
 &\quad (\text{по предположению индукции}) \\
 &= [\theta] \circ [\xi] =
 \end{aligned}$$

(по определению ДС)

$= [\beta]$.

- 2.2. Пусть $\beta \equiv (\theta \cup \xi)$. Т.к. θ является фрагментом β , то согласно предположению индукции, $B2S(\theta)$ использует метки в интервале $[0.. \max B2S(\theta)]$, причём, метка $\max B2S(\theta)$ является её единственной выходной меткой; следовательно, $(B2S(\theta)+1)$ использует метки в интервале $[1.. (\max B2S(\theta)+1)]$, причём, метка $(\max B2S(\theta)+1)$ является её единственной выходной меткой. Аналогично, т.к. ξ является фрагментом β , то согласно предположению индукции, $B2S(\xi)$ использует метки в интервале $[0.. \max B2S(\xi)]$, причём, метка $\max B2S(\xi)$ является её единственной выходной меткой; следовательно, $(B2S(\xi)+\max B2S(\theta)+1)$ использует метки в интервале $[(\max B2S(\theta)+1).. (\max B2S(\theta)+\max B2S(\xi)+1)]$, причём, метка $(\max B2S(\theta)+\max B2S(\xi)+1)$ является её единственной выходной меткой. Поэтому $(B2S(\theta)+1)((\max B2S(\theta)+\max B2S(\xi)+1)/(\max B2S(\theta)+1))$ использует метки в интервале $[1.. \max B2S(\theta)]$ и ещё метку $(\max B2S(\theta)+\max B2S(\xi)+1)$, которая является её единственной выходной меткой. Т.к.

$$\begin{aligned} B2S(\beta) &= \{ (0: \text{if TRUE then } \{1, (\max B2S(\theta)+1)\} \text{ else } \emptyset) \} \cup \\ &\cup (B2S(\theta)+1)((\max B2S(\theta)+\max B2S(\xi)+1)/(\max B2S(\theta)+1)) \cup \\ &\cup (B2S(\xi)+\max B2S(\theta)+1), \end{aligned}$$

то $B2S(\beta)$ использует метки в интервале $[0.. (\max B2S(\theta)+\max B2S(\xi)+1)]$, причём, метка $(\max B2S(\theta)+\max B2S(\xi)+1)$ является её единственной выходной меткой.

Пусть α_β - программа виртуальной HeMo-машины, которая состоит из описаний δ и множества операторов $B2S(\beta)$. В пространстве состояний $SP(\delta)$ имеем:

$$\begin{aligned} [\alpha_\beta] &= \\ &= \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\beta, \text{ которая начинается в конфигурации } (0, s'), \text{ а заканчивается в конфигурации } ((\max B2S(\theta)+\max B2S(\xi)+1), s'') \} = \\ &= \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\theta, \text{ которая начинается в } (0, s') \text{ и заканчивается в } (\max B2S(\theta), s'') \} \cup \\ &\cup \{ (s', s'') : \text{существует конечная полная трасса } \alpha_\xi, \text{ которая начинается в } (0, s') \text{ и заканчивается в } (\max B2S(\xi), s'') \} = \\ &(\text{по предположению индукции}) \\ &= [\theta] \cup [\xi] = \\ &(\text{по определению ДС}) \end{aligned}$$

= $[\beta]$.

2.3. Пусть $\beta \equiv (\theta)^*$. Т.к. θ является фрагментом β , то согласно предположению индукции, $B2S(\theta)$ использует метки в интервале $[0.. \max B2S(\theta)]$, причём, метка $\max B2S(\theta)$ является её единственной выходной меткой; следовательно, $(B2S(\theta)+1)$ использует метки в интервале $[1.. (\max B2S(\theta)+1)]$, причём, метка $(\max B2S(\theta)+1)$ является её единственной выходной меткой. Т.к.

$$B2S(\beta) = \{(0: \text{if TRUE then } \{1, (\max B2S(\theta)+1)\} \text{ else } \emptyset)\} \cup \\ \cup (B2S(\theta)+1)(0/(\max B2S(\theta)+1)),$$

то $B2S(\beta)$ использует метки в интервале $[0.. (\max B2S(\theta)+1)]$, причём, метка $(\max B2S(\theta)+1)$ является её единственной выходной меткой.

Пусть α_β – программа виртуальной НеМо-машины, которая состоит из описаний δ и множества операторов $B2S(\beta)$. В пространстве состояний $SP(\delta)$ имеем:

$$\begin{aligned} [\alpha_\beta] &= \\ &= \{(s', s'') : \text{существует конечная полная трасса } \alpha_\beta, \text{ которая начинается в конфигурации } (0, s'), \text{ а заканчивается в конфигурации } ((\max B2S(\theta)+1), s'')\} = \\ &= \{(s', s'') : \text{существует такие } n \geq 0, \text{ последовательность состояний } s_0, \dots, s_n, \text{ что } s' = s_0, s'' = s_n \text{ и существует конечная полная трасса } \alpha_\beta, \\ &\text{ которая начинается в конфигурации } (0, s'), \text{ заканчивается в конфигурации } ((\max B2S(\theta)+1), s''), \text{ и проходит через конфигурации } (0, s_0), \dots, (0, s_n), \\ &\text{ причём последовательность } s_0, \dots, s_n \text{ - это последовательность всех тех состояний}\} = \\ &= \bigcup_{n \geq 0} \{(s', s'') : \text{существует такая последовательность состояний } s_0, \dots, s_n, \text{ что } s' = s_0, s'' = s_n \text{ и для любого } i \in [0..(n-1)] \text{ существует конечная полная трасса } \alpha_\theta, \text{ которая начинается в } (0, s_i) \text{ и заканчивается в } (\max B2S(\theta), s_{i+1})\} = \\ &= \bigcup_{n \geq 0} \{(s', s'') : \text{существует конечная полная трасса } \alpha_\theta, \text{ которая начинается в } (0, s') \text{ и заканчивается в } (\max B2S(\theta), s'')\}^n = \\ &\text{(по предположению индукции)} \\ &= \bigcup_{n \geq 0} [\theta]^n = \text{(по определению ДС)} = [\beta]. \blacksquare \end{aligned}$$

Утверждение 7

Алгоритм трансляции вычислительных НеМо-программ в вычислительные программы виртуальной НеМо-машины

- является тотально корректным относительно своих пред- и пост- условий,
- имеет квадратичную верхнюю оценку по времени и линейную по памяти (в зависимости от размера дерева грамматического разбора НеМо-программы).

Доказательство (набросок).

Т.к. алгоритм трансляции сводится к тривиальному разбиению программы на описания и тело, а потом вызову процедуры B2S, то тотальная корректность является очевидным следствием из утверждения 1. По аналогичным соображениям оценки сложности алгоритма трансляции сводятся к оценкам сложности процедуры B2S, которую легко доказать индукцией по структуре тела НеМо-программы.

База индукции: B2S осуществляет трансляцию всякого присваивания и теста за фиксированное время, не зависящее от конкретного вида этого присваивания или теста.

Индукционная гипотеза: предположим, что B2S транслирует любое подтело некоторого «составного» тела за квадратичное время на линейной памяти.

Шаг индукции.

B2S осуществляет трансляцию всякого «составного» тела в три этапа:

- 1) выделяет непересекающиеся подтела и конструктор,
- 2) транслирует подтела и модифицирует их метки,
- 3) добавляет оператор, соответствующий конструктору.

Выделение подтел и конструктора осуществляется по дереву грамматического разбора за линейное время и не требует дополнительной памяти. Согласно индукционной гипотезе, трансляция подтел осуществима за квадратичное время на линейной памяти; модификация меток не требует дополнительной памяти, и осуществима за линейное время. Добавление оператора в соответствии с конструктором осуществимо за фиксированное число шагов. ■

*Функциональная схема компиляции вычислительных программ на языке
HeMo*

Определение 8

Компиляция – это преобразование текста грамматически разобранной входной программы (возможно, рассредоточенного по нескольким модулям) в код, исполнимый на объектной машине.

Объектная машина для HeMo – это виртуальная HeMo-машина. Поэтому так называемая функциональная архитектура компиляции синтаксически правильных вычислительных программ на языке HeMo имеет вид, представленный на следующем рисунке, где прямоугольники обозначают функциональные модули, а стрелки – потоки управления и данных.



Остаётся только заметить, что любой реальный компьютер – это не виртуальная HeMo-машина, а поэтому его ассемблер – это не язык программирования виртуальной HeMo-машины. Следовательно, пооператорная интерпретация на языке ассемблера программ на языке виртуальной HeMo-машины может оказаться низкоэффективной и ресурсоёмкой. Поэтому чтобы повысить эффективность реализации и оптимизировать использование ре-

сурсов реальной вычислительной машины может иметь смысл вместо интерпретатора написать транслятор программ виртуальной НеМо-машины в ассемблер (или даже в машинный код). Мы, однако, не будем этим заниматься, т.к. такого рода трансляция уже слишком машинно-зависима и выходит за рамки базового курса по трансляции, статическому анализу и верификации программ.

Пример трансляции

В качестве примера рассмотрим трансляцию вычислительной программы π , для которой уже была построена вычислительная денотационная семантика в лекции 13:

VAR x : INT; VAR y : INT; VAR z : INT;

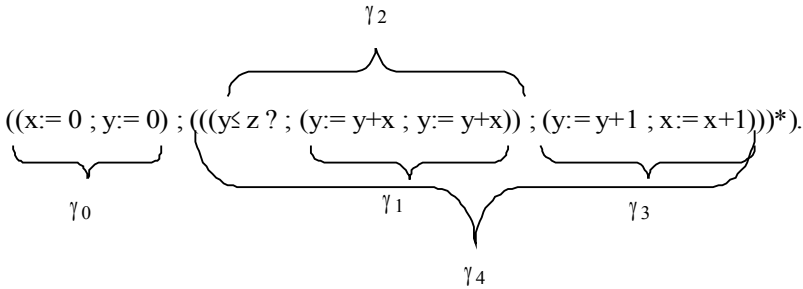
((x:= 0 ; y:= 0) ; (((y≤z ? ; (y:= y+x ; y:= y+x)) ; (y:= y+1 ; x:= x+1))))*).

Совокупность описаний δ программы π – это

VAR x : INT; VAR y : INT; VAR z : INT;

а тело β программы π – это

((x:= 0 ; y:= 0) ; (((y≤z ? ; (y:= y+x ; y:= y+x)) ; (y:= y+1 ; x:= x+1))))*).



Для удобства мы примем следующие обозначения:

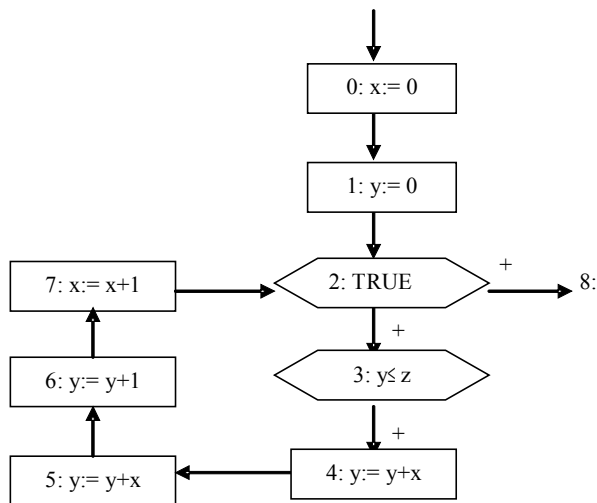
Имеем:

1. $B2S(\beta) = B2S(\gamma_0) \cup (B2S(\gamma_4^*) + \max B2S(\gamma_0))$;
2. $B2S(\gamma_0) = B2S(x:=0) \cup (B2S(y:=0) + \max B2S(x:=0)) =$
 $= \{ (0: x:= 0 \text{ goto } \{1\}) \} \cup (\{ (0: y:= 0 \text{ goto } \{1\}) \} + 1) =$
 $= \{ (0: x:= 0 \text{ goto } \{1\}) \} \cup \{ (1: y:= 0 \text{ goto } \{2\}) \} =$
 $= \{ (0: x:= 0 \text{ goto } \{1\}), (1: y:= 0 \text{ goto } \{2\}) \}$;
3. $B2S(\gamma_4^*) = \{ (0: \text{if TRUE then } \{1, (1 + \max B2S(\gamma_4))\} \text{ else } \emptyset) \} \cup$

- $$\cup (B2S(\gamma_4)+1)(0/(1 + \max B2S(\gamma_4))) ;$$
4. $B2S(\gamma_4) = B2S(\gamma_2) \cup (B2S(\gamma_3) + \max B2S(\gamma_2)) ;$
 5. $B2S(\gamma_2) = B2S(y \leq z ?) \cup (B2S(\gamma_1) + \max B2S(y \leq z ?)) =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset) \} \cup (B2S(\gamma_1) + 1) ;$
 6. $B2S(\gamma_1) = B2S(y := y + x) \cup (B2S(y := y + x) + \max B2S(y := y + x)) =$
 $= \{ (0: y := y + x \text{ goto } \{1\}) \} \cup (\{ (0: y := y + x \text{ goto } \{1\}) \} + 1) =$
 $= \{ (0: y := y + x \text{ goto } \{1\}) \} \cup \{ (1: y := y + x \text{ goto } \{2\}) \} =$
 $= \{ (0: y := y + x \text{ goto } \{1\}), (1: y := y + x \text{ goto } \{2\}) \} ;$
 7. $B2S(\gamma_2) = (\text{в силу 5}) =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset) \} \cup (B2S(\gamma_1) + 1) = (\text{в силу 6}) =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset) \} \cup$
 $\cup \{ (0: y := y + x \text{ goto } \{1\}), (1: y := y + x \text{ goto } \{2\}) \} + 1) =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset) \} \cup$
 $\cup \{ (1: y := y + x \text{ goto } \{2\}), (2: y := y + x \text{ goto } \{3\}) \} =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset),$
 $(1: y := y + x \text{ goto } \{2\}), (2: y := y + x \text{ goto } \{3\}) \} ;$
 8. $B2S(\gamma_3) = B2S(y := y + 1) \cup (B2S(x := x + 1) + \max B2S(y := y + 1)) =$
 $= \{ (0: y := y + 1 \text{ goto } \{1\}) \} \cup (\{ (0: x := x + 1 \text{ goto } \{1\}) \} + 1) =$
 $= \{ (0: y := y + 1 \text{ goto } \{1\}) \} \cup \{ (1: x := x + 1 \text{ goto } \{2\}) \} =$
 $= \{ (0: y := y + 1 \text{ goto } \{1\}), (1: x := x + 1 \text{ goto } \{2\}) \} ;$
 9. $B2S(\gamma_4) = (\text{в силу 4}) = B2S(\gamma_2) \cup (B2S(\gamma_3) + \max B2S(\gamma_2)) =$
 (в силу 7)
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset), (1: y := y + x \text{ goto } \{2\}),$
 $(2: y := y + x \text{ goto } \{3\}) \} \cup (B2S(\gamma_3) + 3) =$
 (в силу 8)
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset), (1: y := y + x \text{ goto } \{2\}),$
 $(2: y := y + x \text{ goto } \{3\}) \} \cup$
 $\cup \{ (0: y := y + 1 \text{ goto } \{1\}), (1: x := x + 1 \text{ goto } \{2\}) \} + 3) =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset), (1: y := y + x \text{ goto } \{2\}),$
 $(2: y := y + x \text{ goto } \{3\}) \} \cup$
 $\cup \{ (3: y := y + 1 \text{ goto } \{4\}), (4: x := x + 1 \text{ goto } \{5\}) \} =$
 $= \{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset), (1: y := y + x \text{ goto } \{2\}),$
 $(2: y := y + x \text{ goto } \{3\}), (3: y := y + 1 \text{ goto } \{4\}), (4: x := x + 1 \text{ goto } \{5\}) \} ;$
 10. $B2S(\gamma_4^*) = (\text{в силу 3}) =$
 $= \{ (0: \text{if TRUE then } \{1, (1 + \max B2S(\gamma_4))\} \text{ else } \emptyset) \} \cup$
 $\cup (B2S(\gamma_4)+1)(0/(1 + \max B2S(\gamma_4))) =$
 (в силу 9)
 $= \{ (0: \text{if TRUE then } \{1, (1 + 5)\} \text{ else } \emptyset) \} \cup$

$$\begin{aligned}
& \cup (\{ (0: \text{if } y \leq z \text{ then } \{1\} \text{ else } \emptyset), (1: y := y+x \text{ goto } \{2\}), \\
& \quad (2: y := y+x \text{ goto } \{3\}), (3: y := y+1 \text{ goto } \{4\}), \\
& \quad (4: x := x+1 \text{ goto } \{5\}) \} + 1) (0/(1+5)) = \\
& = \{ (0: \text{if TRUE then } \{1, 6\} \text{ else } \emptyset) \} \cup \\
& \cup \{ (1: \text{if } y \leq z \text{ then } \{2\} \text{ else } \emptyset), (2: y := y+x \text{ goto } \{3\}), \\
& \quad (3: y := y+x \text{ goto } \{4\}), (4: y := y+1 \text{ goto } \{5\}), \\
& \quad (5: x := x+1 \text{ goto } \{6\}) \} (0/6) = \\
& = \{ (0: \text{if TRUE then } \{1, 6\} \text{ else } \emptyset) \} \cup \\
& \cup \{ (1: \text{if } y \leq z \text{ then } \{2\} \text{ else } \emptyset), (2: y := y+x \text{ goto } \{3\}), \\
& \quad (3: y := y+x \text{ goto } \{4\}), (4: y := y+1 \text{ goto } \{5\}), \\
& \quad (5: x := x+1 \text{ goto } \{0\}) \} = \\
& = \{ (0: \text{if TRUE then } \{1, 6\} \text{ else } \emptyset), (1: \text{if } y \leq z \text{ then } \{2\} \text{ else } \emptyset), \\
& \quad (2: y := y+x \text{ goto } \{3\}), (3: y := y+x \text{ goto } \{4\}), \\
& \quad (4: y := y+1 \text{ goto } \{5\}), (5: x := x+1 \text{ goto } \{0\}) \} ; \\
11. \quad B2S(\beta) = (\text{в силу 1}) = B2S(\gamma_0) \cup (B2S(\gamma_4^*) + \max B2S(\gamma_0)) = \\
& \quad (\text{в силу 2}); \\
& = \{ (0: x := 0 \text{ goto } \{1\}), (1: y := 0 \text{ goto } \{2\}) \} \cup (B2S(\gamma_4^*) + 2) = \\
& \quad (\text{в силу 10}) \\
& = \{ (2: x := 0 \text{ goto } \{1\}), (1: y := 0 \text{ goto } \{2\}) \} \cup \\
& \cup (\{ (0: \text{if TRUE then } \{1, 6\} \text{ else } \emptyset), (1: \text{if } y \leq z \text{ then } \{2\} \text{ else } \emptyset), \\
& \quad (2: y := y+x \text{ goto } \{3\}), (3: y := y+x \text{ goto } \{4\}), \\
& \quad (4: y := y+1 \text{ goto } \{5\}), (5: x := x+1 \text{ goto } \{0\}) \} + 2) = \\
& = \{ (0: x := 0 \text{ goto } \{1\}), (1: y := 0 \text{ goto } \{2\}) \} \cup \\
& \cup \{ (2: \text{if TRUE then } \{3, 8\} \text{ else } \emptyset), (3: \text{if } y \leq z \text{ then } \{4\} \text{ else } \emptyset), \\
& \quad (4: y := y+x \text{ goto } \{5\}), (5: y := y+x \text{ goto } \{6\}), \\
& \quad (6: y := y+1 \text{ goto } \{7\}), (7: x := x+1 \text{ goto } \{2\}) \} = \\
& = \{ (0: x := 0 \text{ goto } \{1\}), (1: y := 0 \text{ goto } \{2\}), \\
& \quad (2: \text{if TRUE then } \{3, 8\} \text{ else } \emptyset), (3: \text{if } y \leq z \text{ then } \{4\} \text{ else } \emptyset), \\
& \quad (4: y := y+x \text{ goto } \{5\}), (5: y := y+x \text{ goto } \{6\}), \\
& \quad (6: y := y+1 \text{ goto } \{7\}), (7: x := x+1 \text{ goto } \{2\}) \} .
\end{aligned}$$

Для того, чтобы наглядно представить получившееся множество операторов, представим его в виде блок-схемы:



Лабораторная работа 4. Трансляция вычислительных HeMo-программ в вычислительные программы виртуальной HeMo-машины

Цель лабораторной работы: генерация по таблице переменных и дереву грамматического разбора синтаксически правильных вычислительных HeMo-программ эквивалентных вычислительных программ виртуальной HeMo-машины.

Что входит в лабораторную работу:

3. Описание алгоритма трансляции вычислительной программы на ВА-ШЕМ диалекте HeMo в семантически эквивалентную вычислительную программу ВАШЕГО диалекта виртуальной HeMo-машины.
4. Реализация транслятора для вычислительных программ на принятом диалекте языка виртуальной HeMo-машины в семантически эквивалентную вычислительную программу виртуальной HeMo-машины.

Этапы:

4. Адаптация общего алгоритма трансляции вычислительных HeMo-программ в эквивалентные вычислительные программы виртуальной

НеМо-машины к принятым диалектам языка НеМо и языка виртуальной НеМо-машины.

5. Дизайн и реализация программы-транслятора вычислительных НеМо-программ в эквивалентные вычислительные программы виртуальной НеМо-машины к принятым диалектам языка НеМо и языка виртуальной НеМо-машины.

Требований «заказчика» к представлению алгоритма трансляции:

Алгоритм должен быть представлен в форме псевдокода или блок-схемы высокого уровня и небольшой сопроводительной записки-комментария об используемых структурах данных, операциях на них, об использовании дерева грамматического разбора. Общий объём описания алгоритма и сопроводительной записки – 1-2 стр. (Блок-схема может быть нарисована от руки.)

Требования «заказчика» к программе-транслятору:

4. Предусловие: Входные данные – таблица переменных и дерево грамматического разбора в том формате, в каком они порождаются программой лексико-синтаксического анализа по исходной программе на ВАШЕМ диалекте языка НеМо.
5. Постусловие: Выходные данные – программа на ВАШЕМ диалекте языка виртуальной НеМо-машины, эквивалентная исходной программе по вычислительной операционной семантике.

Лекция 15. «Универсальный» язык программирования

Алгоритмические свойства программируемых отношений

Определение 9

Пусть задано некоторое множество S . Бинарное отношение R на S называется

1. частично вычислимым (или рекурсивно перечислимым) тогда и только тогда, когда существует алгоритм, который для произвольной пары s' и s'' из S работает следующим образом:
 - 1.1. если $(s', s'') \in R$, то алгоритм завершает свою работу с ответом «ДА»;
 - 1.2. если $(s', s'') \notin R$, то алгоритм не завершает свою работу;
2. вычислимым (или рекурсивным) тогда и только тогда, когда существует алгоритм, который для произвольной пары s' и s'' из S работает следующим образом:

- 2.1. если $(s', s'') \in R$, то алгоритм завершает свою работу с ответом «ДА»,
- 2.2. если $(s', s'') \notin R$, то алгоритм завершает свою работу с ответом «НЕТ».

Всякая синтаксически правильная вычислительная программа на языке НеМо состоит из описаний и тела. Если выбрана семантика наследственно-конечных таблиц над кольцом целых чисел (или любым кольцом вычетов по модулю фиксированного числа) для типов данных, то описания позволяют определить множество состояний, а тело – бинарное отношение вход-выход на этом множестве.

Утверждение 8

Для любой программы α языка НеМо её вычислительная операционная семантика $[\alpha]$ является частично вычислимым бинарным отношением на пространстве состояний $SP(\alpha)$.

Доказательство получается комбинацией утверждения 1 из лекции 14 с утверждением 1 из лекции 12. Пусть α - произвольная вычислительная НеМо-программа, а s' и s'' - произвольная пара состояний из $SP(\alpha)$. Транслируем α в эквивалентную вычислительную программу β виртуальной НеМо-машины и запустим алгоритм обхода расширенного семантического дерева $ET_\beta(s')$ до тех пор пока не достигнем листа вида $(_, s'')$; как только такой лист достигнут, остановимся с ответом «ДА». Имеем: $(s', s'') \in [\alpha]$ тогда и только тогда, когда $ET_\beta(s')$ содержит лист $(_, s'')$. Поэтому, если $(s', s'') \in [\alpha]$, то алгоритм обхода дойдёт до листа $(_, s'')$ и мы получим ответ «ДА», в противном случае мы должны продолжать обход дерева. ■

Возникают две проблемы:

- 1) для всякого ли частично вычислимого бинарного отношения R на множестве SP , которое является пространством состояний какой-либо конечной совокупностью описаний, существует такая НеМо-программа α , что $R = [\alpha]$?
- 2) является ли операционная вычислительная семантика $[\alpha]$ произвольной НеМо-программы α рекурсивным отношением на пространстве состояний $SP(\alpha)$?

Принципиальное затруднение, которое не позволяет сразу ответить на поставленные выше два вопроса, состоит в том, что понятие функции вычислимой алгоритмически принадлежит к числу неопределяемых понятий, оно носит такой же фундаментальный характер как, например, понятие множества. Оба эти понятия нельзя определить через другие, а возможно только постулировать некоторые из их свойств. Постулаты (аксиомы) неформальной теории множеств мы «вспомнили» в 3 лекции, а теперь готовы «вспомнить» основной постулат теории алгоритмов – т. н. «тезис Алонзо Чёрча». Для того, чтобы этот постулат сформулировать, нам придётся ввести некоторые вспомогательное понятие машин М. Минского⁶¹.

В наших терминах машины М. Минского могут быть определены следующим образом.

Определение 10

Машина М. Минского – это любая программа виртуальной НеМо-машины, которая удовлетворяет двум легко проверяемым синтаксическим ограничениям:

- всякая метка метит не более одного оператора, а всякое множество меток после «goto», «then» или «else» состоит ровно из одной метки,
- все переменные имеют целый тип, все выражения в операторах присваивания имеют вид $\langle \text{переменная} \rangle + 1$ или $\langle \text{переменная} \rangle - 1$, а все условия в условных операторах – вид $\langle \text{переменная} \rangle > 0$.

Так как всякая машина М. Минского – это программа виртуальной НеМо-машины, то для каждой такой машины определены понятия пространства состояний, конфигурации, срабатывания, трассы, семантических

⁶¹ Альтернативой могли бы быть, например, машины А. Тьюринга, но ради краткости мы выбираем именно машины М. Минского. В качестве популярного введение в теорию математических машин, формализующих понятия алгоритма, можно порекомендовать книгу Трахтенброт В.А. Алгоритмы и вычислительные автоматы. М., Советское радио, 1974. Конкретно о машинах М. Минского подробнее можно прочитать, например, в книге Котов В.Е. Сети Петри. М., Наука, 1984.

деревьев и вычислительной семантики⁶². Поэтому мы можем определить семантику машин М. Минского следующим образом.

Определение 11

Пусть C – машина М. Минского. Семантика $[C]$ – это как вычислительная семантика $[C]$ программы виртуальной НеМо-машины C .

Таким образом, каждая машина М. Минского – это одна программа для виртуальной НеМо-машины. Разница между виртуальной НеМо-машиной и машиной М. Минского состоит в том, что виртуальная НеМо-машина – это универсальное вычислительное устройство, способное исполнить любую программу на любых входных данных, а машина М. Минского – это специализированное вычислительное устройство, исполняющее только одну программу, но для любых входных данных.

Пусть X и Y – некоторые множества, а $f: X \rightarrow Y$ – (частичная) функция; тогда график функции f – это множество $\{(x, f(x)) : x \in X\}$. В силу того, что в машинах М. Минского всякая метка метит не более одного оператора, а после «goto», «then» или «else» состоит ровно из одной метки, всякая машина М. Минского C является детерминированной в следующем смысле: для любого состояния $s \in SP(C)$ расширенное семантическое дерево $ET_C(s)$ состоит из единственной трассы, которая является или полной, или бесконечной начальной. Поэтому вычислительная семантика $[C]$ любой машины М. Минского C является графиком (частичной) функции $f_C: (INT)^{VAR} \rightarrow (INT)^{VAR}$, где VAR – множество переменных в C , определённой следующим образом:

$$f_C = \lambda s' \in (INT)^{VAR}. \begin{cases} s'' \in (INT)^{VAR}, \text{ если } (s', s'') \in [C], \\ \text{не определено в противном случае.} \end{cases}$$

Утверждение 9

Пусть C – машина М. Минского, а VAR – множество переменных C .

1. График функции f_C является частично вычислимым бинарным отношением на $(INT)^{VAR}$.

⁶² Вообще говоря, для машин М. Минского тип INT интерпретируется только натуральными числами (т.е. всеми неотрицательными целыми числами), причём операция вычитания интерпретируется «вычитанием без перехода через ноль»: $\lambda x. \lambda y. (\text{если } (x-y) > 0 \text{ то } (x-y) \text{ иначе } 0)$. Однако мы будем интерпретировать тип INT обычным образом, т.е. всем множеством целых чисел (что не изменяет вычислительной мощности т.к. у нас под рукой есть предикат «>0»).

2. Если $f_C: (INT)^{VAR} \rightarrow (INT)^{VAR}$ является тотальной функцией, то её график является вычислимым бинарным отношением на $(INT)^{VAR}$.

Доказательство.

Во-первых, согласно утверждению 1, вычислительная семантика любой машины М. Минского является частично вычислимым бинарным отношением на соответствующем пространстве состояний. Однако, этот факт очевиден и без всяких ссылок на утверждение 1, т.к. (в силу детерминированности) обходы расширенных семантических деревьев для машин М. Минского «тривиальны»: достаточно просто последовательно исполнять оператор за оператором в соответствии с передачей управления по «goto», «then» и «else».

Во-вторых, если известно, что f_C является тотальной функцией, то можно предложить следующий алгоритм, который по произвольной паре состояний $s', s'' \in SP(C)$ проверяет принадлежность (s', s'') графику функции f_C :

1. выполнить «обход» расширенного семантического дерева $ET_C(s')$ и найти его единственный лист $(_, s)$;
2. если $s=s''$, то остановиться с ответом «ДА», в противном случае – остановиться с ответом «НЕТ».

В силу предположения о тотальности f_C «шаг» 1 алгоритма рано или поздно завершиться. Корректность алгоритма получается в силу детерминированности машин М. Минского. ■

Определение 12

Пусть VAR – конечное множество переменных, а $f: (INT^{VAR}) \rightarrow (INT^{VAR})$ частичная функция на целых числах $n = |VAR|$ аргументов. Будем говорить, что эта функция реализуется машиной М. Минского C с множеством переменных VAR тогда и только тогда, когда $f = f_C$.

Машины М. Минского и НеМо-программы

Для формулировки следующей пары утверждений нам понадобится дополнительное следующее вспомогательное понятие: пусть X и Y – произвольные множества, а $x \in X$ и $f: X \rightarrow Y$ – произвольные элемент X и (частичная) функция; тогда $f_x: (X \setminus \{x\}) \rightarrow Y$ – это такая (частичная) функция, которая на всех элементах $(X \setminus \{x\})$ совпадает с f . Это понятие, в частности, применимо к состояниям НеМо-программ и программ виртуальной НеМо-машины, которые, как известно являются отображениями из множества переменных в множества значений в соответствии с типами переменных.

Утверждение 10

Существует алгоритм, который по произвольной программе β виртуальной НеМо-машины строит НеМо-программу α_β такую, что совокупность описаний α_β отличается от описаний β только одной дополнительной целой переменной $label$, которой нет в β , и $[\beta] = \{ (s' \setminus_{label}, s'' \setminus_{label}) : s', s'' \in SP(\alpha_\beta) \text{ и } (s', s'') \in [\alpha_\beta] \}$.

Доказательство.

Пусть β – произвольная программа для виртуальной НеМо-машины. Пусть δ – совокупность описаний программы β , а γ – совокупность её операторов, а END – совокупность всех её заключительных меток (которые встречаются в α , но не метят ни одного оператора). Пусть $label$ – новая, не встречающаяся в β переменная.

Для любого оператора присваивания из γ

$k: x := t \text{ goto } L$

(« k », « x », « t » и « L » – произвольные метка, переменная, выражение и конечное множество меток) пусть $\alpha_k: x := t \text{ goto } L$ – это следующее тело НеМо-программы:

$((label = k?) ; x := t ; (\cup_{l \in L} (label := l)))$.

Для любого условного оператора из γ

$k: \text{if } \varphi \text{ then } L^+ \text{ else } L^-$

(« k », « φ », « L^+ » и « L^- » – произвольные метка, условие, и пара конечных множеств меток) пусть $\alpha_k: \text{if } \varphi \text{ then } L^+ \text{ else } L^-$ – это следующее тело НеМо-программы:

$((label = k?) ; ((\varphi? ; (\cup_{l \in L^+} (label := l))) \cup ((\neg\varphi)? ; (\cup_{l \in L^-} (label := l))))$.

Тогда НеМо-программа α_β состоит из совокупности описаний $\delta \cup \{\text{VAR } label : \text{INT}\}$, а её тело – это

$(label := 0) ; (\cup_{op \in \gamma} \alpha_{op})^* ; (\cup_{k \in END} (label = k?))$.

Легко видеть, что для любого оператора $op \in \gamma$ имеем:

$[\alpha_{op}] = \{ ((s' \cup \{(label, k')\}, (s'' \cup \{(label, k'')\})) :$

$s', s'' \in SP(\beta)$, k' и k'' - метки, $((k', s'), (k'', s''))$ – срабатывание оператора op в β }.

Поэтому имеем

$$[\bigcup_{op \in \gamma} \alpha_{op}] = \{ (s' \cup \{(label, k')\}, s'' \cup \{(label, k'')\}) : s', s'' \in SP(\beta), k' \text{ и } k'' - \text{метки}, ((k', s'), (k'', s'')) - \text{срабатывание } \beta \}$$

и, следовательно,

$$\begin{aligned} [(\bigcup_{op \in \gamma} \alpha_{op})^*] &= [(\bigcup_{op \in \gamma} \alpha_{op})]^* = \\ &= \{ (s' \cup \{(label, k')\}, s'' \cup \{(label, k'')\}) : s', s'' \in SP(\beta), k' \text{ и } k'' - \text{метки, и} \\ &\text{существует трасса } \beta, \text{ которая начинается с } (k', s'), \text{ а заканчивается } (k'', s'') \}. \end{aligned}$$

Очевидным образом имеем

$$[label := 0] = \{ (s \cup \{(label, k)\}, s \cup \{(label, 0)\}) : s \in SP(\beta), k \in INT \}$$

и

$$[\bigcup_{k \in END} (label = k?)] = \{ (s \cup \{(label, k)\}, s \cup \{(label, k)\}) : s \in SP(\beta), k \in END \}.$$

Поэтому

$$\begin{aligned} [\alpha_\beta] &= [label := 0] \circ [(\bigcup_{op \in \gamma} \alpha_{op})^*] \circ [\bigcup_{k \in END} (label = k?)] = \\ &= \{ (s' \cup \{(label, k')\}, s'' \cup \{(label, k'')\}) : s', s'' \in SP(\beta), k' = 0 \text{ и } k'' \in END - \end{aligned}$$

метка, и

$$\begin{aligned} &\text{существует трасса } \beta, \text{ которая начинается с } (k', s'), \text{ а заканчивается } (k'', s'') \} = \\ &= \{ (s' \cup \{(label, 0)\}, s'' \cup \{(label, k)\}) : s', s'' \in SP(\beta), k \in END - \text{метки, и } (s', s'') \in [\beta] \}. \end{aligned}$$

$$\text{Отсюда получаем } [\beta] = \{ (s^{\wedge}_{label}, s''^{\wedge}_{label}) : s', s'' \in SP(\alpha_\beta) \text{ и } (s', s'') \in [\alpha_\beta] \}. \blacksquare$$

Из утверждения 3 следует

Утверждение 11

Существует алгоритм, который по произвольной машине М. Минского С строит НеМо-программу α_C такую, что совокупность описаний α_C отличается от описаний С только одной дополнительной целой переменной label, которой нет в С, и $[C] = \{ (s^{\wedge}_{label}, s''^{\wedge}_{label}) : s', s'' \in SP(\alpha_C) \text{ и } (s', s'') \in [\alpha_C] \}$.

Отметим, что утверждения 3 и 4 можно условно назвать леммами о трансляции программ виртуальной НеМо-машины и машин М. Минского в НеМо-программы. Кроме того их можно было бы назвать леммами о трансляции неструктурированных (и детерминированных в случае машин М. Минского) программ в структурированные.

Тезис А. Чёрча и полнота по А. Тьюрингу

Тезис А. Чёрча является постулатом теории алгоритмов. Для машин М. Минского его можно сформулировать следующим образом:

Пусть VAR – конечное множество целочисленных переменных, а $f: \text{INT}^{\text{VAR}} \rightarrow \text{INT}^{\text{VAR}}$ частичная функция на целых числах. График функции f является частично вычислимым отношением тогда и только тогда, когда f реализуется некоторой машиной М. Минского.

В частности, из этого постулата и пункта 2 утверждения 2 следует, что тотальная функция на целых числах имеет вычислимый график тогда и только тогда, когда она реализуется некоторой машиной М. Минского.

Для тех, кто интересуется аргументацией в пользу достоверности тезиса А. Чёрча, отметим, что самым веским аргументом в его пользу является то, что все варианты формализации понятия частичной вычислимости на целых числах, которые были предложены разными математиками в 30-70 годах XX века оказались эквивалентными друг другу и, в частности, машинам М. Минского. Отметим так же, что формализация М. Минского не смотря на свою «программистскую» простоту не является самой известной формализацией, а одной из самых известных формализаций для понятия частичной вычислимости является аппарат машин А. Тьюринга. Кроме того, изначально тезис А. Чёрча был сформулирован не для машин М. Минского, и даже не для машин А. Тьюринга, а для совсем другого формализма для понятия частичной вычислимости на целых числах – т. н. λ -исчисления⁶³.

Утверждение 12

Для любой конечной совокупности описаний переменных δ на языке НеМо, для любого частично вычислимого бинарного отношения $R \subseteq \text{SP}(\delta) \times \text{SP}(\delta)$ существует такая НеМо-программа α_R , которая использует несколько дополнительных целых переменных $\text{new}_1, \dots, \text{new}_n$ (которые не встречаются

⁶³ Можно порекомендовать познакомиться с основами этого формализма по следующей книге Барендрегт Х. Ламбда-исчисление. Его синтаксис и семантика. Гл. 2 – Конверсия, гл. 3 – Редукция, гл. 6 – Классическое ламбда-исчисление. – М., Мир, 1985.

в δ), такая, что $R = \{ ((s^{\wedge}_{new1} \dots \wedge_{newn}), (s''^{\wedge}_{new1} \dots \wedge_{newn})) : s', s'' \in SP(\alpha_R) \text{ и } (s', s'') \in [\alpha_R] \}$.

Доказательств (набросок).

Чтобы избежать обсуждения несущественных для данного курса деталей доказательства как-то

- использования нескольких переменных,
- кодировки целыми числами наследственно-конечных таблиц,
- предположим, что δ состоит из единственного описания некоторой фиксированной целой переменной: $VAR\ x : INT$. При таких предположениях любое состояние $SP(DE)=INT^{\{x\}}$ удобно отождествить с числом – значением переменной x в этом состоянии.

Т.к. R – частично вычислимое отношение, то существует алгоритм, который для произвольной пары значений k' и k'' переменной x работает следующим образом:

1. если $(k', k'') \in R$, то алгоритм завершает свою работу с ответом «ДА»,
2. если $(k', k'') \notin R$, то алгоритм не завершает свою работу.

Пусть ans (от «answer»), bef (от «before») aft (от «after») – новые целочисленные переменные. Рассмотрим функцию $f: INT^{\{bef, aft, ans\}} \rightarrow INT^{\{bef, aft, ans\}}$, определённую следующим образом: если значение k' переменной bef и значение k'' переменной aft находятся в отношении R , то не изменяя этих значений присвоить переменной ans значение 1.

График функции f является частично вычислимым, т.к. R – частично вычислимое отношение. В силу тезиса А. Чёрча, f реализуется некоторой машиной М. Минского $C: f = f_C$, т.е. график f – это семантика машины $[C]$. Согласно утверждению 4, существует такая НеМо-программа α_C , что совокупность описаний α_C отличается от описаний C только одной дополнительной целой переменной $label$, которой нет в C , и $[C] = \{ (s^{\wedge}_{label}, s''^{\wedge}_{label}) : s', s'' \in SP(\alpha_C) \text{ и } (s', s'') \in [\alpha_C] \}$. По построению НеМо-программы α_C , её совокупность описаний состоит из декларации четырёх целочисленных переменных bef, aft, ans и $label$. Пусть β_C – тело программы α_C .

Теперь рассмотрим следующую НеМо-программу α_R :

$VAR\ x, bef, aft, ans, label : INT;$
 $(bef := x ; (aft := aft+1 \cup aft := aft-1)^* ; \beta_C ; (ans=1?) ; x:=aft) .$

По построению эта программа

- 1) сначала принимает начальное значение фиксированной переменной x в качестве значения переменной bef (оператор $bef:=x$);
- 2) затем недетерминированно генерирует произвольное значение для переменной aft (тело $(aft:=aft+1 \cup aft:=aft-1)^*$);
- 3) потом исполняет тело β_C программы α_C , в котором значения переменных x , bef и aft не изменяются, а значение переменной ans становится 1 тогда и только тогда, когда значения переменных bef и aft связаны отношением R ;
- 4) в заключении переменной x присваивается значение переменной aft , если $ans=1$ (тело $(ans=1?) ; x:=aft$).

Поэтому имеем:

$$R = \{ ((s^{\backslash label \backslash bef \backslash aft \backslash ans}), (s'^{\backslash label \back bef \back aft \back ans}) : s', s'' \in SP(\alpha_R) \text{ и } (s', s'') \in [\alpha_R] \}$$

что и требовалось доказать. ■

Определение 13

Язык программирования называется универсальным алгоритмическим языком, если

- целые числа представимы (или могут быть промоделированы) в терминах типов данных этого языка,
- любая его программа реализует некоторый «вычислительный» алгоритм,
- любой «вычислительный» алгоритм над любыми типами данных этого языка реализуем в виде программы на этом языке.

Иногда вместо «универсальный алгоритмический» говорят «полный по Тьюрингу», так как машины А. Тьюринга являются одним из самых общепринятых формализмов для понятия алгоритма.

НеМо является универсальным алгоритмическим языком программирования т.к. (согласно определению вычислительной операционной семантики) целые числа являются разрешённым типом данных языка, и (согласно утверждениям 1 и 5) для любой конечной совокупности описаний переменных δ на НеМо, для любого бинарного отношения $R \subseteq SP(\delta) \times SP(\delta)$ имеем: R является частично вычислимым тогда и только тогда, когда существует такая НеМо-программа α , которая использует несколько вспомогательных переменных new_1, \dots, new_n , что $R = \{ ((s^{\backslash new_1 \dots \back new_n}), (s'^{\back new_1 \dots \back new_n}) : s', s'' \in SP($

α) и $(s', s'') \in [\alpha]\}$. Тем самым мы показали, что вычислительные HeMo-программы могут служить формализмом для понятия алгоритм на типах данных языка HeMo и, одновременно, ответили на первый вопрос, поставленный в начале лекции. Ответ на второй вопрос уже не входит в рамки настоящего курса, а скорее относится к общей теории алгоритмов. Заинтересованным читателям можно порекомендовать, например, обратиться к следующим книгам:

1. Ершов Ю.Л., Палютин Е.А., Математическая логика. Гл. 7 Алгоритмы и рекурсивные функции. М., Наука, 1987.
2. Трахтенброт В.А. Алгоритмы и вычислительные автоматы. М., Советское радио, 1974.

Детерминированное программирование на HeMo

Семантика вычислительных HeMo-программ – это бинарные отношения на пространстве состояний. Но среди всех бинарных отношений особое место принадлежит графикам (частичных) функций, т.е. таким отношениям R , для которых выполнено следующее свойство: для любых значений a, b и c , если $(a, b) \in R$ и $(a, c) \in R$, то $b = c$. Особая роль графикам функций принадлежит в силу возможности однозначно (детерминированно) определить пару (если она вообще существует) по первому элементу.

Определение 14

Вычислительная программа называется детерминированной, если её операционная семантика – это график некоторой (частичной) функции.

Однако семантика HeMo-программ в общем случае – это частично вычислимые бинарные отношения, но необязательно графики функций. Причина по которой так получилось, простая: вместо обычных детерминированных программных конструкторов выбора «IF-THEN-ELSE» и итерации «WHILE-DO» язык HeMo использует недетерминированные « \cup » и « $*$ ». Естественная неформальная семантика этих новых конструкций следующая:

- в IF-THEN-ELSE выполняется первое тело, если условие верно, иначе – второе;
- в WHILE-DO тело выполняется пока условие верно.

В лекции 1 уже обсуждалось на неформальном уровне как детерминированные конструкции выразить через недетерминированные в вычислительных программах:

- IF ϕ THEN α ELSE β эквивалентно $((\phi? ; \alpha) \cup ((-\phi)? ; \beta))$,

- WHILE ϕ DO α эквивалентно $((\phi? ; \alpha)^* ; (\neg\phi)?)$.

Теперь мы можем формализовать эти неформальные эквивалентности.

Для этого расширим язык HeMo детерминированными конструкторами «IF-THEN-ELSE» и «WHILE-DO» следующим образом. Во-первых, изменим определение понятия $\langle \text{тело} \rangle$ (см. лекцию 9)

$\langle \text{тело} \rangle ::= \langle \text{присваивание} \rangle \mid \langle \text{тест} \rangle \mid (\langle \text{тело} \rangle ; \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle \cup \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle)^*$

на следующее определение:

$\langle \text{тело} \rangle ::= \langle \text{присваивание} \rangle \mid \langle \text{тест} \rangle \mid (\langle \text{тело} \rangle ; \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle \cup \langle \text{тело} \rangle) \mid (\langle \text{тело} \rangle)^* \mid$
 $(\text{IF } \langle \text{отношение} \rangle \text{ THEN } \langle \text{тело} \rangle \text{ ELSE } \langle \text{тело} \rangle) \mid$
 $(\text{WHILE } \langle \text{отношение} \rangle \text{ DO } \langle \text{тело} \rangle)$

Во-вторых, при выбранной и зафиксированной совокупности описаний и соответствующем пространстве состояний SP доопределим традиционную операционную семантику вход-выход для тела HeMo-программ на SP (см. лекцию 13) следующим образом (которые соответствуют естественной неформальной семантике):

- детерминированный выбор: для любых состояний s' и s'' , для любого условия ϕ , для любых тел α и β имеем: $s' \langle (\text{IF } \phi \text{ THEN } \alpha \text{ ELSE } \beta) \rangle s'' \Leftrightarrow$ если условие ϕ выполнено в состоянии s' , то $s' \langle \alpha \rangle s''$, иначе $s' \langle \beta \rangle s''$;
- детерминированная итерация: для любых состояний s' и s'' , для любого условия ϕ , для любого тела α имеем: $s' \langle (\text{WHILE } \phi \text{ DO } \alpha) \rangle s'' \Leftrightarrow$ существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$, условие ϕ невыполнено в s_n , и $s_i \langle \alpha \rangle s_{i+1}$ и условие ϕ выполнено в s_i для всех $0 \leq i < n$.

Утверждение 13

Пусть SP – пространство состояний. Тогда

- для любого условия ϕ , для любых тел α и β традиционная операционная семантика вычислительных программ $(\text{IF } \phi \text{ THEN } \alpha \text{ ELSE } \beta)$ и $((\phi? ; \alpha) \cup ((\neg\phi)? ; \beta))$ совпадают;

- для любого условия ϕ , для любого тела α традиционная операционная семантика вычислительных программ ($\text{WHILE } \phi \text{ DO } \alpha$) и $((\phi? ; \alpha)^* ; (\neg \phi)?)$ совпадают.

Доказательство. Пусть s' и s'' - произвольные состояния из SP. В первом случае имеем: $s' \langle \text{IF } \phi \text{ THEN } \alpha \text{ ELSE } \beta \rangle s'' \Leftrightarrow$ если ϕ выполнено в s' , то $s' \langle \alpha \rangle s''$, иначе $s' \langle \beta \rangle s'' \Leftrightarrow \phi$ выполнено в s' и $s' \langle \alpha \rangle s''$, или ϕ невыполнено в s' и $s' \langle \beta \rangle s'' \Leftrightarrow s' \langle \phi? ; \alpha \rangle s''$ или $s' \langle (\neg \phi)? ; \beta \rangle s'' \Leftrightarrow s' \langle (\phi? ; \alpha) \cup ((\neg \phi)? ; \beta) \rangle s''$.

Во втором случае имеем: $s' \langle \text{WHILE } \phi \text{ DO } \alpha \rangle s'' \Leftrightarrow$

\Leftrightarrow существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$, ϕ невыполнено в s_n , а $s_i \langle \alpha \rangle s_{i+1}$ и ϕ выполнено в s_i для всех $0 \leq i < n \Leftrightarrow$

\Leftrightarrow существует такое $n \geq 0$ и последовательность состояний s_0, \dots, s_n , что $s_0 = s'$, $s_n = s''$, $s_n \langle (\neg \phi)? \rangle s_n$, и $s_i \langle \alpha ; \phi? \rangle s_{i+1}$ для всех $0 \leq i < n \Leftrightarrow s'' \langle (\neg \phi)? \rangle s''$ и $s' \langle (\alpha ; \phi?)^* \rangle s'' \Leftrightarrow s' \langle (\alpha ; \phi?)^* ; (\neg \phi)? \rangle s''$. ■

Таким образом, мы можем добавить обе конструкции IF-THEN-ELSE и WHILE-DO в язык HeMo не увеличивая вычислительной мощности языка а как некий «синтаксический сахар» для удобства программирования знакомых алгоритмов привычными средствами⁶⁴. Но при этом естественно возникает класс всех таких HeMo-программ, которые написаны с использованием конструкций IF-THEN-ELSE и WHILE-DO без (явного) использования базовых конструкций « \cup » и « $*$ ». Мы будем называть программ псевдодетерминированными⁶⁵.

Утверждение 14

Все псевдодетерминированные вычислительные HeMo-программы являются детерминированными.

Доказательство.

Выберем произвольно и зафиксируем псевдодетерминированную программу π . Пусть δ - её совокупность описаний, а α - её тело. Следовательно,

⁶⁴ При этом пока можно считать, что обе эти конструкции – это макросы, которые перед трансляцией (т.е. во время прецессирования) будут заменяться на эквивалентные базовые конструкции.

⁶⁵ В следующей лекции 16 мы обсудим почему в названии этого класса программ присутствует приставка «псевдо».

пространство состояний $SP=SP(\delta)$ тоже выбрано и зафиксировано. Доказательство проведём индукцией по структуре тела программы α .

База индукции: тело состоит из одного присваивания или одного теста. Оба случая тривиально следуют из определения семантики присваивания и теста, поэтому рассмотрим только случай присваивания. Пусть это присваивание « $x:=t$ ». Имеем: $[x:=t] = \{ (s', s'') : s'' = \text{UPD}(s', x, s'(t)) \}$, т.е. $[x:=t]$ – это график функции $\lambda s \in SP. \text{UPD}(s, x, s(t))$.

Индукционная гипотеза: предположим, что утверждение верно для всех тел, которые являются фрагментами тела α .

Шаг индукции: α – это или последовательная композиция, или детерминированный выбор, или детерминированная итерация.

Последовательная композиция. Пусть α есть $(\beta ; \gamma)$. Тогда имеем:

$[\alpha] = \{ (s', s'') : \text{существует такое состояние } s, \text{ что } (s', s) \in [\beta] \text{ и } (s, s'') \in [\gamma] \}$.

Пусть s', s''_1 и s''_2 три произвольных состояния такие, что $(s', s''_1) \in [\alpha]$ и $(s', s''_2) \in [\alpha]$. Тогда должны существовать состояния s_1 и s_2 такие, что $(s', s_1) \in [\beta]$, $(s_1, s''_1) \in [\gamma]$ и $(s', s_2) \in [\beta]$, $(s_2, s''_2) \in [\gamma]$. По предположению индукции для β имеем $s_1=s_2$. Следовательно $s''_1=s''_2$ уже по предположению индукции для γ .

Детерминированный выбор. Пусть α есть $(\text{IF } \phi \text{ THEN } \beta \text{ ELSE } \gamma)$. Тогда имеем:

$[\alpha] = \{ (s', s'') : \text{если в } s' \text{ выполнено } \phi, \text{ то } (s', s'') \in [\beta], \text{ иначе } (s', s'') \in [\gamma] \}$.

Пусть s', s''_1 и s''_2 три произвольных состояния такие, что $(s', s''_1) \in [\alpha]$ и $(s', s''_2) \in [\alpha]$. Если ϕ выполнено в s' , то $(s', s''_1) \in [\beta]$, $(s', s''_2) \in [\beta]$ и, следовательно $s''_1=s''_2$ по предположению индукции для β . Если ϕ невыполнено в s' , то $(s', s''_1) \in [\gamma]$, $(s', s''_2) \in [\gamma]$ и, следовательно $s''_1=s''_2$ уже по предположению индукции для γ .

Детерминированная итерация. Пусть α есть $(\text{WHILE } \phi \text{ DO } \beta)$. Тогда имеем:

$[\alpha] = \{ (s', s'') : \text{существует такое } n \geq 0 \text{ и } s_0, \dots, s_n, \text{ что } s'=s_0, s''=s_n, \text{ в } s_n \text{ невыполнено } \phi, (s_i, s_{i+1}) \in [\beta] \text{ и в } s_i \text{ выполнено } \phi \text{ для всех } 0 \leq i < n \}$.

Пусть s', s''_a и s''_b три произвольных состояния такие, что $(s', s''_a) \in [\alpha]$ и $(s', s''_b) \in [\alpha]$. Тогда пусть n_1 и n_2 соответствующие числа, a, a_0, \dots, a_{n_1} и b_0, \dots, b_{n_2} такие последовательности состояний, что

– $s'=a_0, s''_a=a_{n_1}$, в a_{n_1} невыполнено ϕ , $(a_i, a_{i+1}) \in [\beta]$ и в a_i выполнено ϕ для всех $0 \leq i < n_1$;

- $s'_b=b_0$, $s''_b=b_{n2}$, в b_{n2} невыполнено φ , $(b_i, b_{i+1}) \in [\beta]$ и в b_i выполнено φ для всех $0 \leq i < n2$.

Так-как $s'_a=a_0$, и $s'_b=b_0$, то $a_0=b_0$ и по предположению индукции для β имеем $a_1=b_1$. Так-как $a_1=b_1$, то по предположению индукции для β имеем $a_2=b_2$. Продолжая таким образом (т.е. индукцией по i), получаем: $n1=n2$ и $a_0=b_0$, $a_1=b_1, \dots, a_{n1}=b_{n2}$. Но $s''_a=a_{n1}$ и $s''_b=b_{n2}$, поэтому $s''_a=s''_b$. ■

Ook!

Согласно свободной интернет-энциклопедии Wikipedia⁶⁶, термин «эзотерические языки программирования» сложился в среде высококлассных программистов-системщиков (хакеров в хорошем смысле слова) для обозначения языков, спроектированных как некий курьёз или как средство задавать очень специфических алгоритмов. Практичность таких языков не является целью авторов эзотерических языков, несмотря на возможность задавать алгоритмы (то есть программировать) на таких языках и несмотря на полноту по А. Тьюрингу большинства из таких языков. Большинство таких языков получается в результате отказа от общепринятой нотации, стилей и/или эклектическим сочетанием разных парадигм программирования. Поэтому для непосвящённого программиста текст программы на таком языке – это просто совершенно бессмысленный набор символов, в котором невозможно ухватиться ни за какую знакомую конструкцию. В качестве примера эзотерического языка программирования познакомимся с неформальным описанием языка программирования для орангутангов Ook!⁶⁷.

Язык разработал Д. Морган-Мар (David Morgan-Mar). Основополагающий принцип языка Ook! – это простота лексем, доступная даже для произношения орангутангами. Поэтому язык имеет только три лексемы:

- «Ook.»,
- «Ook?»,
- «Ook!».

Язык использует «большой» одномерный неиндексированный массив, каждая ячейка которого способна хранить любое целое число. В качестве возможного «большого» массива рекомендуется выбрать массив из 5, 10, или 20 ячеек (по числу пальцев одной руки, двух рук или всех пальцев рук и ног). В начальный момент вычислений все ячейки массива инициализированы

⁶⁶ Wikipedia – см. URL http://en.wikipedia.org/wiki/Main_Page, статья Esoteric programming language – см. URL http://en.wikipedia.org/wiki/Esoteric_programming_language, статья List of esoteric programming languages – см. URL http://en.wikipedia.org/wiki/List_of_esoteric_programming_languages.

⁶⁷ Ook! – см. URL <http://www.dangermouse.net/esoteric/ook.html>.

нулями. Кроме того, имеется указатель, который может перемещаться по массиву от ячейки к соседям в соответствии с командами, описанными ниже. В начальный момент указатель установлен на самой левой ячейке. Таким образом, архитектура виртуальной Ook!-машины может быть изображена примерно следующим образом (здесь «большой» массив состоит из 10 ячеек):



Программы на языке Ook! – это произвольные конечные последовательности команд, в которых «Ook? Ook!» и «Ook! Ook?» являются парными открывающей и закрывающей скобками для блоков (и, одновременно, командами условного перехода). Язык Ook! имеет следующий набор команд:

- «Ook. Ook?» - сдвигает (если возможно) указатель на одну ячейку вправо;
- «Ook? Ook.» - сдвигает (если возможно) указатель на одну ячейку влево;
- «Ook. Ook.» - увеличивает на 1 число в ячейке, на которой установлен указатель;
- «Ook! Ook!» - уменьшает на 1 число в ячейке, на которой установлен указатель;
- «Ook. Ook!» - читает очередной символ из стандартного потока ввода и заносит его ASCII-код в ячейку, на которой установлен указатель;
- «Ook! Ook.» - выводит в стандартный поток вывода символ, ASCII-код которого хранится в ячейке, на которой установлен указатель;
- «Ook! Ook?» – если в ячейке, на которой установлен указатель, хранится 0 (нуль), то передаёт управление команде, которая следует за соответствующей «Ook? Ook!»;
- «Ook? Ook!» – если в ячейке, на которой установлен указатель, хранится не нуль, то передаёт управление команде, которая следует за соответствующей «Ook! Ook?».

Если верить официальному сайту языка, то существует несколько реализаций Ook! В частности, Л. Пит (Lawrence Pit) реализовал Ook!-компилятор для .Net, который называется (разумеется!) Ook#. Мы же ограничимся

только неформальным утверждением, что язык Ook! Полон по А. Тьюрингу. Для неформального обоснования этого утверждения ограничимся замечанием, что фактически Ook! – это структурированный диалект языка машин М. Минского, в котором число используемых переменных совпадает с размером массива.

Часть IV. На пути к верификации

Лекция 16. Определение аксиоматической семантики вычислительных программ (на примере языка HeMo)

Зачем нужна аксиоматическая семантика вычислительных программ

В лекции 13 уже были описаны несколько эквивалентных семантик для вычислительных программ на языке HeMo, как-то традиционная операционная семантика (ТОС), структурная операционная семантика (СОС) и денотационная семантика (ДС). Все эти семантики имеют дело с состояниями (парами состояний, последовательностями состояний, множествами пар состояний), то есть определяется непосредственно на семантическом уровне.

Наоборот, аксиоматическая семантика вычислительных программ определяется сначала на синтаксическом уровне в виде аксиоматической системы в терминах *формальных троек Хоара для языка HeMo* (или, для краткости, просто *троек*). Неформально говоря, тройки – это утверждения о частичной корректности, каждая тройка состоит из программы и двух формул – предусловия и постусловия (см. лекцию 1). Но, строго говоря, тройки – новый синтаксический объект, поэтому они сами нуждаются в формальном определении синтаксиса и семантики. Поэтому в этой лекции мы определим синтаксис троек, зададим аксиоматическую систему, прокомментируем неформальную семантику правил вывода и аксиом этой системы в терминах частичной корректности вычислительных программ, а в следующей лекции определим формальную семантику троек в терминах истинности утверждений о частичной корректности.

Как показано в лекции 13 (см. утверждения 1 и 3), связь всех трех ранее определенных семантик вычислительных программ (ТОС, СОС и ДС) исключительно проста: все три семантики определяют одно и то же бинарное отношение вход-выход на пространстве состояний. Аксиоматическая семантика уже не определяет бинарного отношения вход-выход на пространстве состояний. Поэтому связь аксиоматической семантики вычислительных программ и ранее определенных семантик ТОС, СОС и ДС будет установлена утверждениями о надежности и полноте. Суть этих утверждений состоит в том, что всякая доказуемая тройка является истинным утверждением о частичной корректности, а всякое истинное утверждение о частичной корректности является доказуемым. Но цель и смысл аксиоматической се-

мантики – как раз доказывать истинность утверждений о частичной корректности по возможности не прибегая к семантике программ.

Так как всякая тройка – это условие частичной корректности, то она имеет следующий вид: $\{\varphi\}\alpha\{\psi\}$, где α – вычислительная программа, а φ и ψ – формулы для предусловия и постусловия. Естественно, что в качестве вычислительных программ мы будем использовать программы на языке НеМо, а в качестве пред- и пост- условий – формулы математической логики. Формальные синтаксис и семантика языка НеМо-программ уже были определены в лекциях 9 и 13, а синтаксис и семантика языка формул математической логики – в лекции 3 соответственно. Однако на данный момент эти определения не согласованы, так сказать «не подогнаны» друг к другу. Например, определение переменных, которые используются в программе, и переменных, которые используются в формулах, не согласованы, так как одни – типизированные, а другие – нет. Не согласовано также, что такое символы отношений и символы операций, и как они соотносятся с понятиями $\langle \text{символ_отношения} \rangle$ и $\langle \text{знак_операции} \rangle$. Поэтому нам необходимо формально и согласованно переопределить язык формул и язык программ⁶⁸. Следующие определения фиксируют вариант, которые будут использоваться в курсе.

Синтаксис аксиоматической семантики языка НеМо

Определение⁶⁹ контекстно-свободного синтаксиса формальных троек Хоара для языка НеМо состоит из нескольких «смысловых» блоков: ядра, общих понятий, программ, (пред- и пост-) условий (и типизированных формул или предложений).

⁶⁸ Эта необходимость согласовать данные ранее конкретные определения путем их переопределения не является методической ошибкой курса, а как раз является методическим приемом построения курса. Этот прием, во-первых, отражает реальную историческую диалектику развития отрасли, которая называется «теорией языков программирования», когда разные понятия из разных разделов математики при применении к конкретному языку программирования должны быть конкретизированы и согласованы. А во-вторых, этот прием позволяет не перегружать изначально конкретные определения деталями, которые понадобятся только через 6-10 лекций, и которые можно будет легко ввести в нужном месте пользуясь аналогией с уже освоенными понятиями.

⁶⁹ Как и раньше в лекции 9 будем использовать в нотации Бэкуса-Наура символ « \sim » для обозначения пробела. Кроме того, для избегания путаницы с метаскобками « $\{$ » и « $\}$ » для обозначения открывающей и закрывающей фигурных скобок будем использовать подчеркнутые символы « $\{$ » и « $\}$ ». Разумеется, что «на письме» символ пробела, отображаются непосредственно, а фигурные скобки – без всяких подчеркиваний.

Ядро нашего варианта контекстно-свободного синтаксиса языка троек состоит из единственного определения:

$$\langle \text{тройка} \rangle ::= \{ \sim \langle \text{предусловие} \rangle \sim \{ \sim \langle \text{программа} \rangle \sim \{ \sim \langle \text{постусловие} \rangle \sim \{$$

Блок общих определений включает несколько уже нам знакомых по лекции 9 понятий, которые используются как в определении программ, так и в (пред- и пост-) условий. Это понятия $\langle \text{тип} \rangle$, $\langle \text{описание} \rangle$, $\langle \text{операнд} \rangle$, $\langle \text{знак_операции} \rangle$, $\langle \text{символ_отношения} \rangle$, $\langle \text{число} \rangle$ и $\langle \text{переменная} \rangle$, их точные определения даны в лекции 9.

Определение понятия $\langle \text{программа} \rangle$ также совпадает с определением из лекции 9.

Блок определений понятий (пред- и пост-) условий и типизированных формул конкретизирует общее определение языка математической логики из лекции 3.

$$\langle \text{типизированная формула} \rangle ::= \langle \text{параметры} \rangle \langle \text{формула} \rangle$$

$$\langle \text{параметры} \rangle ::= \{ \langle \text{описание} \rangle ; \sim \}$$

$$\langle \text{предусловие} \rangle ::= \langle \text{типизированная формула} \rangle$$

$$\langle \text{постусловие} \rangle ::= \langle \text{формула} \rangle$$

$$\begin{aligned} \langle \text{формула} \rangle ::= & \text{TRUE} \mid \text{FALSE} \mid \\ & \mid \langle \text{элементарная формула} \rangle \mid \neg (\langle \text{формула} \rangle) \mid \\ & \mid (\langle \text{формула} \rangle \wedge \langle \text{формула} \rangle) \mid (\langle \text{формула} \rangle \vee \langle \text{формула} \rangle) \mid \\ & \mid \forall \langle \text{описание} \rangle (\langle \text{формула} \rangle) \mid \exists \langle \text{описание} \rangle (\langle \text{формула} \rangle) \end{aligned}$$

$$\langle \text{элементарная формула} \rangle ::= \langle \text{терм} \rangle \langle \text{символ_отношения} \rangle \langle \text{терм} \rangle$$

$$\langle \text{терм} \rangle ::= \langle \text{операнд} \rangle \mid \langle \text{арифметический_терм} \rangle \mid \langle \text{функциональный_терм} \rangle$$

$$\langle \text{арифметический_терм} \rangle ::= (\langle \text{терм} \rangle \langle \text{знак_операции} \rangle \langle \text{терм} \rangle)$$

$$\langle \text{функциональный_терм} \rangle ::= \text{APP}(\langle \text{терм} \rangle, \sim \langle \text{терм} \rangle) \mid \text{UPD}(\langle \text{терм} \rangle, \sim \langle \text{терм} \rangle, \sim \langle \text{терм} \rangle)$$

На этом определение контекстно-свободного синтаксиса троек Хоара для языка НеМо завершается.

Определение. Список контекстных зависимостей для формализованного варианта языка троек обобщает определение контекстных зависимостей для НеМо-программ из лекции 9. Он состоит из трёх ограничений, аналогичных уже рассмотренным для НеМо-программ, и одного нового дополнительного ограничения. Любая синтаксически правильная тройка, (типизированная) формула, программа должна удовлетворять ниже перечисленным четырем контекстным ограничениям:

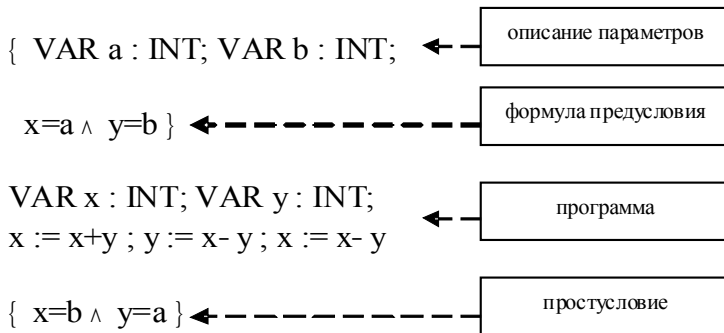
- всякая встречающаяся переменная должна иметь единственное описание в пределах тройки;
- тип левой и правой части любого отношения должен быть INT;
- типы левой и правой части любого присваивания должны совпадать;
- в левых частях присваиваний могут стоять только программные переменные, а в правых — только свободные и программные переменные.

На этом определение синтаксиса формальных троек Хоара для языка НеМо полностью завершено. Но прежде чем определить собственно аксиоматическую семантику, стоит прокомментировать неформальный смысл последнего контекстного ограничения. Оно позволяет изменять значения программных переменных (т.е. переменных описанных в программе) только в программе посредством явного присваивания нового значения, перебирать значения связанных переменных (то есть переменных описанных в подкванторных выражениях) только в формулах кванторами, и запрещает как-либо варьировать значения свободных переменных (то есть параметров, или, точнее, переменных описанных в параметрах). Заметим так же, что первое контекстное ограничение позволяет избежать коллизии программных, связанных и свободных переменных: «роли» переменных не меняются и, кроме того, одна и та же связанная переменная не может связываться кванторами в разных местах. В частности, следующая «формула» $\forall \text{VAR } x : \text{INT}; (x > 3) \wedge \forall \text{VAR } x : \text{INT}; (x > 3)$ не является синтаксически правильной «формулой», но может быть переписана корректно, например, следующим образом:

$$\forall \text{VAR } x : \text{INT}; (x > 3) \wedge \forall \text{VAR } y : \text{INT}; (y > 3) .$$

Приведем три примера синтаксически правильных троек.

Пример 1: перестановка значений целочисленных переменных местами.



В этом примере переменные «a» и «b» – параметры или свободные переменные, переменные «x» и «y» – программные переменные, а связанных переменных нет. Пример хорошо иллюстрирует, зачем нужны параметры: оставаясь неизменными, они позволяют ссылаться на начальные значения переменных.

Пример 2: наибольший общий делитель двух положительных целых чисел a и b.

```

{ VAR a : INT; VAR b : INT; a>0 ∧ b>0 }
VAR x : INT; VAR y : INT;
x := a ; y := b ;
(( x>y ? ; x := x-y ) ∪ ( x<y ? ; y := y-x ))* ;
x=y ?
{ (∃ VAR m : INT (m>0 ∧ a=m×x) ) ∧ (∃ VAR n : INT (n>0 ∧ b=n×x) ) ∧
  ∀ VAR z : INT
  ( (∃ VAR p : INT (p>0 ∧ a=p×z) ) ∧ (∃ VAR q : INT (q>0 ∧ b=q×z)) → z≤x) }
  
```

В этом примере уже используются кванторы и связанные переменные «m», «n», «p», «q» и «z», причем, в соответствии с контекстными ограничениями, каждая связанная переменная имеет ровно одно описание и только одну подформулу, в которой она используется.

Этот пример уже несколько сложнее первого, и поэтому заслуживает комментария. Неформально этот пример можно было бы переписать так:


```

{ а и b – целые положительные числа }
VAR x : INT; VAR y : INT;
x := a ; y := b ; WHILE x≠y DO IF x<y THEN x := x-y ELSE y := y-x
{ x – наибольший общий делитель чисел а и b }

```

Во-первых, для программ легко обосновать эквивалентность с точки зрения их вычислительной семантики используя утверждение 6 из лекции 15:

$$\begin{aligned}
& [((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x))^* ; x = y ?] = \\
& = [(x = y ? ; ((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x))^* ; x = y ?] = \\
& = [(x = y ? ; ((x > y ? ; x := x - y) \cup (x \leq y ? ; y := y - x))^* ; x = y ?] = \\
& = [(x \neq y ? ; \text{IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x)^* ; x = y ?] = \\
& = [\text{WHILE } x \neq y \text{ DO IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x].
\end{aligned}$$

В постусловии тоже легко «узнать» определение понятия «наибольший общий делитель»:

- VAR m : INT (m > 0 ∧ a = m × x) означает, что «x делит a»,
- VAR n : INT (n > 0 ∧ b = n × x) означает, что «x делит b»,
- VAR z : INT ((∃ VAR p : INT (p > 0 ∧ a = p × z)) ∧ (∃ VAR q : INT (q > 0 ∧ b = q × z)) → z ≤ x) означает, что «любой делитель а и b не больше x».

Тогда почему наш пример 2 такой громоздкий? – Главная причина в том, что в языке формальных троек нет специального синтаксического обозначения для понятия «наибольший общий делитель», а потому мы должны были явно сформулировать его определение пользуясь возможностями синтаксиса троек.

Пример 3: сумма первых m элементов целочисленного массива r.

```

{ VAR m : INT; VAR r : (INT ARRAY OF INT);
  VAR sigma : ((INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT));
  m ≥ 0 ∧
  ∀ VAR f : (INT ARRAY OF INT)
    ( APP(APP(sigma,f),0)=0 ∧
      ∀ VAR n : INT ( n > 0 → APP(APP(sigma,f),n)=
        APP(APP(sigma,f),n-1)+APP(f,n) ) ) }

```

```

VAR k : INT; VAR t : INT;
k := 1 ; t := 0 ;
( k ≤ m ? ; t := t+APP(r,k) ; k:= k+1 ) * ; k > m ?
{ t=APP(APP(sigma,r),m) }

```

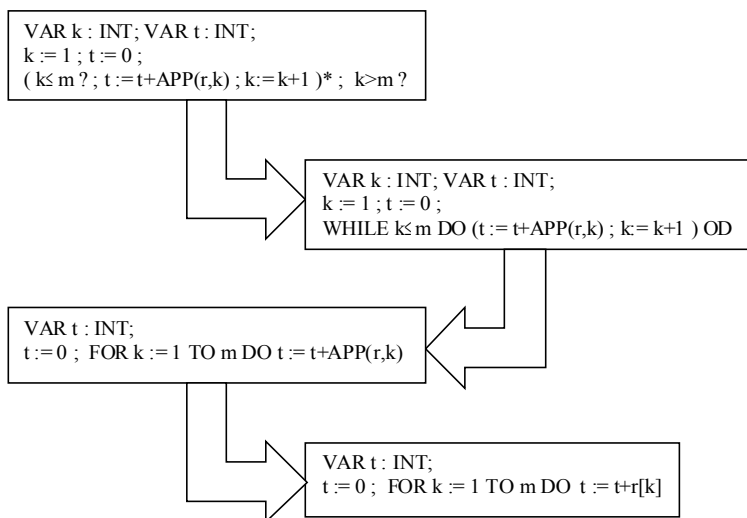
Остановимся подробнее на этом примере. На неформальном языке его можно переписать например так:

```

{ m целое положительное число, а r – целочисленный массив }
VAR t : INT;
t := 0 ; FOR k := 1 TO m DO t := t+r[k]
{ t=Σk∈[1..m]r[k] }

```

Синтаксически правильную и неформальную программы в данном случае легко «отождествить»



и «понять» параметры m и r . Значительно труднее «понять» параметр s и «отождествить» предусловия и постусловия в формальной тройке и в неформальной. Дело здесь обстоит следующим образом. Неформальное постусловие «утверждает», что $t = \sum_{k \in [1..m]} r[k]$. В этом условии знак Σ выступает в качестве функции двух аргументов

$$\Sigma \equiv \lambda r \in \text{INT}^{\text{INT}}. \lambda m \in \text{INT}. (r(1) + \dots + r(m)).$$

Но в языке троек нет такой функции, а есть только обычные бинарные арифметические операции $+$, $-$, \times , $/$ и функциональные операции APP и UPD. Поэтому пришлось явно ввести эту функцию sigma, объявив ее параметром, и аксиоматизировать ее в предусловии:

$$\begin{aligned} &\forall \text{VAR } f : (\text{INT ARRAY OF INT}) \\ &(\text{APP}(\text{APP}(\text{sigma}, f), 0) = 0 \wedge \\ &\quad \forall \text{VAR } n : \text{INT} (n > 0 \rightarrow \text{APP}(\text{APP}(\text{sigma}, f), n) = \\ &\quad \quad \text{APP}(\text{APP}(\text{sigma}, f), n-1) + \text{APP}(f, n))) \end{aligned}$$

Эта аксиоматизация в полуформальной нотации выглядела бы так:

$$\begin{aligned} &\forall f \in \text{INT}^{\text{INT}} (\Sigma_{k \in [1..0]} f(k) = 0 \wedge \\ &\quad \forall n \in \text{INT} (n > 0 \rightarrow \Sigma_{k \in [1..n]} f(k) = \Sigma_{k \in [1..(n-1)]} f(k) + f(n))) \end{aligned}$$

Последний пример может породить некоторое недоумение: почему язык формальных троек такой «тяжеловесный» по сравнению с полуформальным языком? – Дело в том, что полуформальный язык допускает множество «умолчаний» и «смешение» синтаксических и семантических сущностей. А формальный язык должен использовать только явно представленные сущности и разделяет синтаксис и семантику. Поэтому синтаксически правильная тройка и получилась такой «громоздкой».

Определение аксиоматической семантики HeMo

Определение аксиоматической семантики языка HeMo использует следующие синтаксические метAPERменные:

- ρ, δ – для описаний переменных,
- ϕ, ψ, ξ – для формул,
- α, β – для тел программ,
- x и t – для переменной и выражения.

Тогда аксиоматическая семантика языка HeMo – это следующая аксиоматическая система AS:

Аксиома A для присваивания (Assignment)

$$\{ \rho \ \xi \ v_x \} \delta \ (x := t) \{ \xi \}$$

($\xi \ v_x$ обозначает результат замены всех вхождений переменной x в ξ на терм t)

Аксиома T для теста (Test)

$$\{ \rho \ (\phi \rightarrow \psi) \} \delta \ (\phi ?) \{ \psi \}$$

Правило PS/CW усиления посылки и ослабления заключения
(Premise Strengthening/Conclusion Weakening)

$$\rho \delta (\varphi \rightarrow \iota) \quad \{ \rho \iota \} \delta \alpha \{ \xi \} \quad \rho \delta (\xi \rightarrow \psi)$$

$$\{ \rho \varphi \} \delta \alpha \{ \psi \}$$

Правило SC последовательной композиции (Sequential Composition)

$$\{ \rho \varphi \} \delta \alpha \{ \xi \} \quad \{ \rho \xi \} \delta \beta \{ \psi \}$$

$$\{ \rho \varphi \} \delta (\alpha ; \beta) \{ \psi \}$$

Правило NC недетерминированного выбора (Nondeterministic Choice)

$$\{ \rho \varphi \} \delta \alpha \{ \psi \} \quad \{ \rho \varphi \} \delta \beta \{ \psi \}$$

$$\{ \rho \varphi \} \delta (\alpha \cup \beta) \{ \psi \}$$

Правило LI инварианта цикла (Loop Invariant)

$$\{ \rho \iota \} \delta \alpha \{ \iota \}$$

$$\{ \rho \iota \} \delta (\alpha *) \{ \iota \}$$

Следует заметить, что определенная выше аксиоматическая семантика языка НеМо синтаксически корректна в следующем смысле.

Утверждение 15

Для любого правила вывода системы AS кроме аксиом, если все посылки правила – синтаксически правильные (типизированные) формулы и/или тройки, то и заключение правила – синтаксически правильная тройка.

Доказательство этого утверждения рутинно. Достаточно заметить, что

- (типизированные) формулы в заключении «наследуются» из посылок,
- тело программы в заключении получается из тел программ в посылках,
- описания переменных в заключении не дублируются.

■

Представление системы AS в определении чересчур громоздко. «Сэкономить» можно на том, что совокупности описаний свободных и программных переменных (ρ и δ) не изменяются системой AS. Поэтому при построении дерева вывода в системе AS обе совокупности описаний можно не представлять в дереве, а зафиксировать и вынести «за кадр». Следователь-

но, саму систему AS удобнее сразу представить в виде, в котором описаний свободных и программных переменных не присутствуют явно, но предполагается, что они синтаксически совместны с используемыми формулами и программами. В таком представлении система AS приобретает следующий вид:

Правило PS/CW	Правило LI
$\frac{\varphi \rightarrow \{t\} \langle \xi \rangle \quad \xi \rightarrow \mu}{\{ \langle \varphi \rangle \langle t \rangle \}}$	$\frac{\{t\} \langle t \rangle}{\{t\} \langle \xi \rangle \{t\}}$
Правило SC	Правило NC
$\frac{\{ \langle \varphi \rangle \langle \xi \rangle \} \quad \{ \xi \} \{ \beta \} \{ \mu \}}{\{ \langle \varphi \rangle \langle \alpha \rangle \{ \beta \} \{ \mu \} \}}$	$\frac{\{ \langle \varphi \rangle \langle \mu \rangle \} \quad \{ \langle \varphi \rangle \{ \beta \} \{ \mu \} \}}{\{ \langle \varphi \rangle \langle \alpha \rangle \{ \beta \} \{ \mu \} \}}$
Аксиома* A	Аксиома T
$\{ \xi_\alpha \} \langle x := t \rangle \{ \xi \}$	$\{ \langle \varphi \rightarrow \mu \rangle \} \{ \langle \varphi \rangle \{ \mu \} \}$

(* : ξ_α обозначает результат замены всех вхождений переменной x в ξ на терм t . Заметим, что в силу принятых нами контекстных ограничений на язык троек, все вхождения программных переменных являются свободными в предусловии и постусловии.)

Приведем пример дерева вывода в системе AS для синтаксически правильной троек из примера 1. В этом случае вынесенная и фиксированная совокупность описаний следующая:

VAR a : INT, VAR b : INT, VAR x : INT, VAR y : INT.

Тогда возможное дерево вывода можно представить следующим образом:



Заметим, что это дерево вывода не является доказательством, так как в нем есть лист $(x=a \wedge y=b) \rightarrow ((x+y) - ((x+y) - y) = b \wedge (x+y) - y = a)$, который не является аксиомой. Более того, это дерево не может быть «дополнено» до доказательства, так как в системе AS нет правил вывода для (типизированных) формул.

Неформальная семантика аксиоматической семантики НеМо

Неформально говоря, аксиоматическая семантика предназначена для доказательства утверждений о частичной корректности. Поэтому эта семантика обязательно должна быть надежной, то есть всякая доказуемая тройка должна быть истинной. Желательно также, что бы аксиоматическая семантика была полной, то есть всякая истинная тройка могла бы быть доказанной.

Убедимся сначала на неформальном уровне, что аксиоматическая семантика языка НеМо является надежной. При этом будем пользоваться неформальным определением истинности условий частичной корректности (см. лекцию 1) и неформальным понятием, что такое истинная формула.

Надежность аксиомы A. Всякий оператор присваивания « $x := t$ » в языке НеМо имеет только одно явное действие – сделать значение переменной « x » равным значению выражения « t », и не имеет никакого иного побочного эффекта. Поэтому равносильно, проверять ли истинность какого-либо ξ после выполнения этого присваивания « $x := t$ », или просто подставить явно это выражение « t » вместо « x » в ξ и проверить истинность получившегося $\xi_{t/x}$. Например, тройка (с опущенными описаниями)

$$\{ 3=y \} (x := 3) \{ x=y \}$$

естественно является истинной: все равно, что $x=y$, когда значение «х» стало 3, или что проверить $3=y$.

Надежность аксиомы Т. Всякий тест « ϕ ?» в языке НеМо не изменяет значений переменных, а только «разрешает» продолжить вычисления в случае, когда ϕ истинна. Поэтому если известно, что имеет место $(\phi \rightarrow \psi)$, и вычисление программы $(\phi ?)$ успешно завершилось, то естественно, после завершения этой программы имеет место ψ . Например, тройки (с опущенными описаниями)

$$\begin{aligned} &\{ x=3 \rightarrow x>2 \} (x=3 ?) \{ x>2 \} \\ &\{ x=3 \rightarrow x=0 \} (x=3 ?) \{ x=0 \} \end{aligned}$$

обе является истинными. Естественно, что первая тройка не вызывает никаких сомнений. А вот вторая выглядит несколько парадоксально на первый взгляд! Но на самом деле ситуация подобна ситуации с импликацией, когда из лжи следует все что угодно: если уж верно что из равенства $x=3$ следует равенство $x=0$, и тест « $x=3$?» успешно проверен, то также верно что $x=0$.

Надежность правила PS/CW. Предположим, что обе формулы $(\phi \rightarrow \iota)$ и $(\xi \rightarrow \psi)$ и тройка $\{ \iota \} \alpha \{ \xi \}$ истинны. В силу истинности $(\phi \rightarrow \iota)$, если перед вычислением программы α имеет место ϕ , то имеет место и свойство ι . В силу истинности $\{ \iota \} \alpha \{ \xi \}$, если это вычисление завершается, то завершается оно так, что имеет место свойство ξ . В силу истинности $(\xi \rightarrow \psi)$, в этом заключительном состоянии должно иметь место ψ . – Следовательно, тройка $\{ \phi \} \alpha \{ \psi \}$ тоже истинна. Таким образом, истинность посылок в правиле PS/CW влечет истинность заключения.

Надежность правила SC. Предположим, что обе тройки $\{ \phi \} \alpha \{ \xi \}$ и $\{ \xi \} \beta \{ \psi \}$ истинны. Пусть перед вычислением программы $(\alpha ; \beta)$ имеет место ϕ . Так как язык НеМо структурирован (то есть в программах нет переходов «GOTO» из одной части в другую), то в этом вычислении есть момент, когда закончилось вычисление по программе α , и сразу начинаются и идут до конца вычисления по программе β . Так как перед вычислением имело место свойство ϕ , то в момент после вычислений по программе α (но перед вычислениями по программе β) имеет место свойство ξ (в силу истинности $\{ \phi \} \alpha \{ \xi \}$). Так как в момент перед вычислениями по программе β (но после вычислений по программе α) имеет место свойство ξ , то после вычислений

по программе β имеет место свойство ψ (в силу истинности $\{\xi\}\beta\{\psi\}$). – Следовательно, тройка $\{\phi\}(\alpha;\beta)\{\psi\}$ тоже истинна. Таким образом, истинность посылок в правиле SC влечет истинность заключения. Следующий пример (с опущенными описаниями) иллюстрирует суть приведенного рассуждения:

$$\frac{\{x=3\}(x := x+2)\{x=5\} \quad \{x=5\}(x:= 4+x)\{x=9\}}{\{x=3\}(x := x+2 ; x:= 4+x)\{x=9\}}$$

Вычисление по программе $(x := x+2 ; x:= 4+x)$, начинающееся с истинности $x=3$, – это три последовательных значения 3, 5 и 9, которые принимает переменная x . Момент, когда закончилось вычисление по программе α , и сразу начинаются и идут до конца вычисления по программе β , – это момент, когда переменная x только что приняла значение 5.

Надежность правила NC. Предположим, что обе тройки $\{\phi\}\alpha\{\psi\}$ и $\{\phi\}\beta\{\psi\}$ истинны. Пусть перед вычислением программы $(\alpha \cup \beta)$ имеет место ϕ . Так как язык НеМо структурирован, то в этом вычислении не происходит «перескоков» из программы α в программу β . Поэтому это вычисление или является вычислением программы α , или является вычислением программы β . Для определенности пусть это вычисление программы α . Так как перед вычислением имело место свойство ϕ , то после вычислений по программе α имеет место ψ (в силу истинности $\{\phi\}\alpha\{\psi\}$). – Следовательно, тройка $\{\phi\}(\alpha\cup\beta)\{\psi\}$ тоже истинна. Таким образом, истинность посылок в правиле NC влечет истинность заключения. Следующий пример (с опущенными описаниями) иллюстрирует суть приведенного рассуждения:

$$\frac{\{x=3\}(x := x+2)\{x>2\} \quad \{x=3\}(x:= 4+x)\{x>2\}}{\{x=3\}(x := x+2 \cup x:= 4+x)\{x>2\}}$$

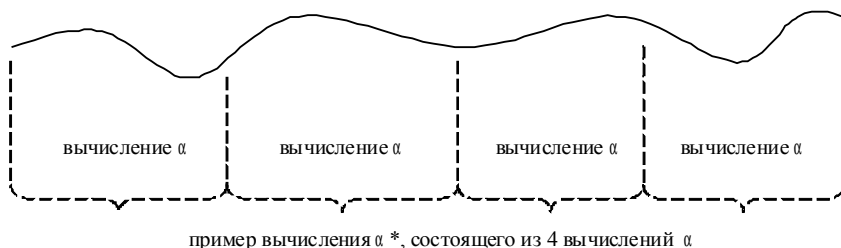
Программа $(x := x+2 \cup x:= 4+x)$ имеет всего два вычисления, которые начинаются с истинности $x=3$: эти вычисления представляются двумя последовательностями значений переменной x , первая из них 3, 5, а вторая 3, 7. Первая из этих последовательностей – единственное вычисление программы $x:=x+2$, которое начинается с истинности $x=3$, а вторая – единственное вычисление программы $x:=x+4$, которые начинаются с истинности

$x=3$. В обоих случаях после вычислений имеет место $x>2$. Поэтому $x>2$ имеет место после любого вычисления ($x := x+2 \cup x := 4+x$), которое начинается с истинности $x=3$.

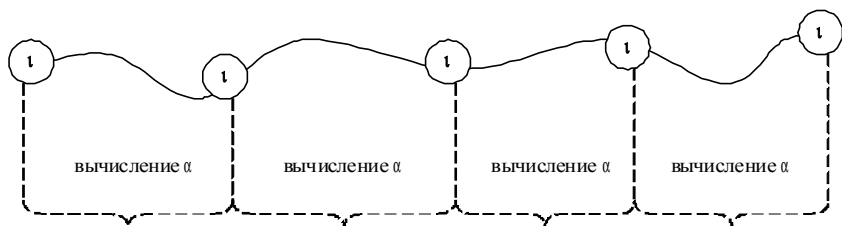
Надежность правила LI. Предположим, что тройка $\{1\}\alpha\{1\}$ истинна. Пусть перед вычислением программы α^* имеет место 1 . Но так как α^* – это недетерминированное количество итераций программы α (0 раз, 1 раз, 2 раза, и так далее), то всякое конкретное вычисление α^* является вычислением программы

$$\underbrace{(\alpha ; \dots ; \alpha)}_{n\text{-раз}}$$

для некоторого $n \geq 0$, т.е. «распадается» на несколько вычислений по программе α (подобно тому как распадалось вычисление по программе $(\alpha ; \beta)$ на вычисление по α , а потом по β в правиле SC):



Если вычисление по α^* состояло из 0 итераций α , то состояние не изменилось, и истинность 1 тоже. Если же вычисление по α^* содержало $n > 0$ итераций α , то в силу предположения об истинности 1 перед началом вычислений по α^* , заключаем, что 1 имеет место перед первой итерацией 1 . В силу предположения об истинности тройки $\{1\}\alpha\{1\}$, 1 имеет место перед началом второй итерации (то есть в момент окончания первой итерации α); поэтому в силу предположения об истинности тройки $\{1\}\alpha\{1\}$, 1 имеет место перед началом третьей итерации (то есть в момент окончания второй итерации α), и т.д. столько раз, какого n . Таким образом, после каждой очередной итерации α во время вычисления по α^* , имеет место 1 .



пример вычисления α^* , состоящего из 4 вычислений α

В частности, свойство t имеет место в после последнего вычисления, т.е. после всего вычисления по программе α^* . Поэтому тройка $\{t\}\alpha^*\{t\}$ истинна. Таким образом, истинность посылок в правиле LI влечет истинность заключения.

Примеры вывода в аксиоматической семантике

Построим возможные варианты вывода⁷⁰ в системе AS для двух оставшихся нерассмотренными примеров синтаксически корректных троек.

Пример 2. В этом случае вынесенная и фиксированная совокупность описаний следующая:

$\text{VAR } a : \text{INT}, \text{VAR } b : \text{INT}, \text{VAR } x : \text{INT}, \text{VAR } y : \text{INT}.$

Для удобства представления дерева вывода, во-первых, обозначим через α следующее тело программы:

$(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x),$

а во-вторых, примем следующие обозначения для формул:

- $\text{pos}(p, q)$ для $(p > 0 \wedge q > 0)$, выражающую что p и q имеют позитивные значения;
- $\text{div}(p, q, r)$ для $\exists \text{VAR } m : \text{INT} (m > 0 \wedge p = m \times r) \wedge (\exists \text{VAR } n : \text{INT} (n > 0 \wedge q = n \times r))$, выражающую, что значение r является общим делителем значений p и q ;
- $\text{les}(p, q, r)$ для $\forall \text{VAR } z : \text{INT} ((\exists \text{VAR } p : \text{INT} (p > 0 \wedge p = p \times z)) \wedge (\exists \text{VAR } q : \text{INT} (q > 0 \wedge q = q \times z))) \rightarrow z \leq r$, выражающую, что все делители значений p и q не превосходят значения r .

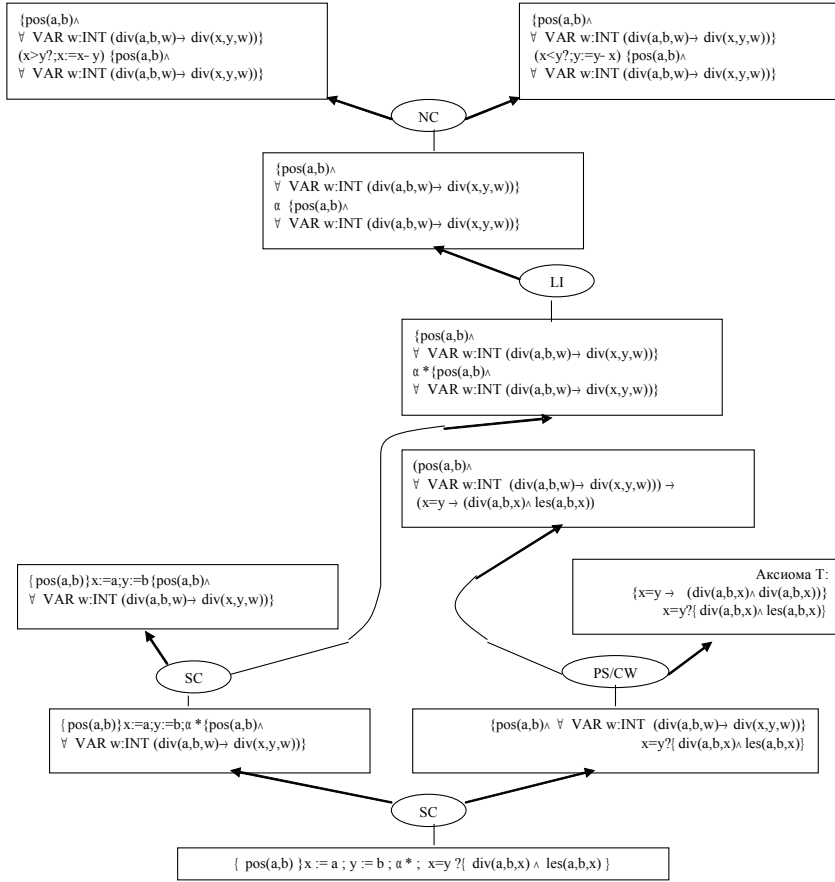
Кроме того, при подстановке на место какой-либо переменной s из p, q и r какого-либо выражения t в формулы $\text{div}(p, q, r)$ и $\text{les}(p, q, r)$ вместо $(\text{div}(p, q, r))_{t/s}$ и $(\text{les}(p, q, r))_{t/s}$ будем использовать явную подстановку, например, $\text{div}(a, b, x - y)$ вместо $\text{div}(a, b, x)_{(x-y)/x}$.

⁷⁰ Мы сейчас не обсуждаем, как вывод был построен, а только обсуждаем его корректность. Методика построения вывода в системе AS будет рассмотрена в следующих лекциях.

Тогда пример 2 принимает следующий сравнительно компактный вид:
 $\{ \text{pos}(a,b) \} x := a ; y := b ; \alpha^* ; x=y ? \{ \text{div}(a,b,x) \wedge \text{les}(a,b,x) \}$

Возможное дерево вывода в системе AS для этой тройки приведено на следующей странице. Обратите внимание, что это дерево (также как и дерево в примере 1) «неполное» так как в нем есть листья, которые не являются аксиомами. Но дерево вывода из первого примера нельзя «нарастить вверх», так как все его листья являются аксиомами или типизированной формулой (а для типизированных формул в системе AS нет правил вывода). А в данном примере дерево возможно «нарастить вверх» до дерева, в котором все листья или аксиомы, или типизированные формулы. Однако для того, что бы это сделать, нам удобнее будет использовать линейный формат представления деревьев доказательства, – менее наглядный для человека, но более пригодный для представления в компьютере. Этот формат уже был формально определен в лекции 4, но до сих пор не использовался.

Для аксиоматической семантики линейное представление какого-либо дерева вывода (или просто линейный вывод) – это конечная последовательность нумерованных строк, каждая из которых соответствует единственному узлу дерева и содержит тройку, соответствующий этому узлу, и список номеров членов этой же последовательности, которые соответствуют наследникам этого узла в дереве, причем, если m -ый член этой последовательности ссылается на номера m_1, \dots, m_n , то $m > m_1, \dots, m_n$. Для удобства проверки вывода мы будем включать в каждую строчку линейного вывода название соответствующего правила системы AS. Ниже приведен возможный вариант линейного вывода для дерева вывода, приведенного на предыдущей странице.



1. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} (x>y ? ; x := x-y) \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$
2. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} (x<y ? ; y := y-x) \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$
3. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} \alpha \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, из 1 и 2 по правилу NC
4. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} \alpha^* \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, из 3 по правилу LI

5. $\{ \text{pos}(a,b) \} x := a ; y := b \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$
6. $\{ \text{pos}(a,b) \} x := a ; y := b ; \alpha^* \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, из 4 и 5 по правилу SC
7. $(\text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w))) \rightarrow (x=y \rightarrow (\text{div}(a,b,x) \wedge \text{les}(a,b,x)))$
8. $\{ x=y \rightarrow (\text{div}(a,b,x) \wedge \text{div}(a,b,x)) \} x=y ? \{ \text{div}(a,b,x) \wedge \text{les}(a,b,x) \}$, аксиома T
9. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} x=y ? \{ \text{div}(a,b,x) \wedge \text{les}(a,b,x) \}$, из 7 и 8 по правилу PS/CW
10. $\{ \text{pos}(a,b) \} x := a ; y := b ; \alpha^* ; x=y ? \{ \text{div}(a,b,x) \wedge \text{les}(a,b,x) \}$, из 6 и 9 по правилу SC

В представленном линейном выводе не имеют «обоснования» (то есть вывода) в терминах системы AS строчки 1, 2, 5 и 7. Строчка 7 содержит типизированную формулу, и поэтому не может быть снабжена каким-либо дополнительным обоснованием в AS. Но строчки 1, 2 и 5 содержат тройки, которые могут быть обоснованы в системе AS. Ниже приведены варианты вывода⁷¹ троек, содержащихся в этих строчках.

- a. $(\text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w))) \rightarrow (x > y \rightarrow \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}((x-y),y,w)))$
 - b. $\{ x > y \rightarrow \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}((x-y),y,w)) \} x > y ? \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}((x-y),y,w)) \}$, аксиома T
 - c. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} x > y ? \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}((x-y),y,w)) \}$, из (A) и (B) по правилу PS/CW
 - d. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}((x-y),y,w)) \} x := x-y \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, аксиома A
 - e. (2 из предыдущего вывода)
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} (x > y ? ; x := x-y) \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$,
из (C) и (D) по правилу SC
- A. $(\text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w))) \rightarrow (x < y \rightarrow \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,(y-x),w)))$

⁷¹ В этих вариантах использованы буквенные и римские номера строчек для того, что бы сохранить преемственность в нумерации строк вывода, приведенного выше.

- B. $\{ x < y \rightarrow \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,(y-x),w)) \} x < y ?$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,(y-x),w)) \}$, аксиома T
- C. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} x < y ?$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,(y-x),w)) \}$,
из (A) и (B) по правилу PS/CW
- D. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,(y-x),w)) \} y := y-x$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, аксиома A
- E. (2 из предыдущего вывода)
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \} (x < y ? ; y := y-x)$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$,
из (C) и (D) по правилу SC
- I. $\text{pos}(a,b) \rightarrow (\text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(a,b,w)))$
- II. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(a,b,w)) \} x := a$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,b,w)) \}$, аксиома A
- III. $\{ \text{pos}(a,b) \} x := a \{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,b,w))$
 $\}$, из (I) и (II) по правилу PS/CW
- IV. $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,b,w)) \} y := b$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$, аксиома A
- V. (5 из предыдущего вывода)
 $\text{pos}(a,b) \} x := a ; y := b$
 $\{ \text{pos}(a,b) \wedge \forall \text{VAR } w : \text{INT} (\text{div}(a,b,w) \rightarrow \text{div}(x,y,w)) \}$,
из (III) и (IV) по правилу SC

Таким образом, чтобы получить вариант «полного» вывода тройки из примера 2 в системе AS, можно объединить все четыре приведенных вывода и расставить их, например, следующим образом: (a), (b), (c), (d), (A), (B), (C), (D), (I), (II), (III), (IV), (1 заменить на e), (2 заменить на E), (3), (4), (5 заменить на V), (6), (7), (8), (9) и (10). Получившийся линейный вывод будет полным в системе AS, будет «почти доказательством» (так как содержит необоснованные типизированные формулы).

Пример 3. В этом случае вынесенная и фиксированная совокупность описаний следующая:

VAR m : INT; VAR r : (INT ARRAY OF INT);
VAR sigma : ((INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT));
VAR k : INT; VAR t : INT;

Для удобства представления вывода, во-первых, обозначим через $(A\Sigma)$ следующую формулу

$$\begin{aligned} & \forall \text{ VAR } f : (\text{INT ARRAY OF INT}) \\ & (\text{APP}(\text{APP}(\text{sigma},f),0)=0 \wedge \\ & \quad \forall \text{ VAR } n : \text{INT } (n>0 \rightarrow \\ & \quad \quad \text{APP}(\text{APP}(\text{sigma},f),n)=\text{APP}(\text{APP}(\text{sigma},f),n-1)+\text{APP}(f,n))) \end{aligned}$$

и, во-вторых, обозначим через α следующее тело программы:

$$k \leq m ? ; t := t + \text{APP}(r,k) ; k := k + 1$$

Тогда тройка из примера 3 примет следующий вид:

$$\{ m \geq 0 \wedge (A\Sigma) \} k := 1 ; t := 0 ; \alpha^* ; k > m ? \{ t = \text{APP}(\text{APP}(\text{sigma},r),m) \}$$

Возможный вывод этой тройки следует:

1. $\{ m \geq 0 \wedge (A\Sigma) \wedge 1 \geq 1 \wedge 1 \leq (m+1) \wedge 0 = \text{APP}(\text{APP}(\text{sigma},r), (1-1)) \} k := 1$
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge 0 = \text{APP}(\text{APP}(\text{sigma},r), (k-1)) \}$
2. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge 0 = \text{APP}(\text{APP}(\text{sigma},r), (k-1)) \} t := 0$
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma},r), (k-1)) \}$
3. $\{ m \geq 0 \wedge (A\Sigma) \wedge 1 \geq 1 \wedge 1 \leq (m+1) \wedge 0 = \text{APP}(\text{APP}(\text{sigma},r), (1-1)) \} k := 1 ; t$
 $:= 0$ $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq$
 $1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma},r), (k-1)) \}$, из 1 и 2 по
 правилу SC
4. $(m \geq 0 \wedge (A\Sigma)) \rightarrow$
 $(m \geq 0 \wedge (A\Sigma) \wedge 1 \geq 1 \wedge 1 \leq (m+1) \wedge 0 = \text{APP}(\text{APP}(\text{sigma},r), (1-1)))$
5. $\{ m \geq 0 \wedge (A\Sigma) \} k := 1 ; t := 0$
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma},r), (k-1)) \}$,
 из 4 и 3 по правилу PS/CW
6. $(m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma},r), (k-1))) \rightarrow$
 $(k \leq m \rightarrow m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge$
 $(t + \text{APP}(r,k)) = \text{APP}(\text{APP}(\text{sigma},r), ((k+1)-1)))$
7. $\{ k \leq m \rightarrow m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge$
 $(t + \text{APP}(r,k)) = \text{APP}(\text{APP}(\text{sigma},r), ((k+1)-1)) \}$
 $k \leq m ?$

- $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge$
 $(t + \text{APP}(r, k)) = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) - 1)) \}$, аксиома T
8. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$
 $k \leq m$?
 $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge$
 $(t + \text{APP}(r, k)) = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) - 1)) \}$,
из 6 и 7 по правилу PS/CW
9. $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge$
 $(t + \text{APP}(r, k)) = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) - 1)) \}$ $t := t + \text{APP}(r, k)$
 $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) -$
 $1)) \}$, аксиома A
10. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$
 $k \leq m$? ; $t := t + \text{APP}(r, k)$
 $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) -$
 $1)) \}$, из 8 и 9 по правилу SC
11. $\{ m \geq 0 \wedge (A\Sigma) \wedge (k+1) \geq 1 \wedge (k+1) \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), ((k+1) -$
 $1)) \}$ $k := k + 1$ $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k$
 $- 1)) \}$, аксиома A
12. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$ α
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$,
из 10 и 11 по правилу SC
13. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$ α^*
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$,
из 12 по правилу LI
14. $\{ m \geq 0 \wedge (A\Sigma) \}$ $k := 1$; $t := 0$; α^*
 $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$,
из 5 и 13 по правилу SC
15. $(m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1))) \rightarrow$
 $(k > m \rightarrow k > m \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)))$
16. $\{ k > m \rightarrow k > m \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$ $k > m$?
 $\{ k > m \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$, аксиома T
17. $(k > m \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1))) \rightarrow$
 $t = \text{APP}(\text{APP}(\text{sigma}, r), m)$
18. $\{ m \geq 0 \wedge (A\Sigma) \wedge k \geq 1 \wedge k \leq (m+1) \wedge t = \text{APP}(\text{APP}(\text{sigma}, r), (k-1)) \}$ $k > m$?
 $\{ t = \text{APP}(\text{APP}(\text{sigma}, r), m) \}$, из 15, 16 и 17 по правилу PS/CW
19. $\{ m \geq 0 \wedge (A\Sigma) \}$ $k := 1$; $t := 0$; α^* ; $k > m$?
 $\{ t = \text{APP}(\text{APP}(\text{sigma}, r), m) \}$, из 14 и 18 по правилу SC

Данный пример является «полным» в аксиоматической семантике, так как в нем остались недоказанными только типизированные формулы.

Лабораторная работа 5. Грамматический разбор и внутреннее представление формальных троек Хоара для языка НеМо

Цель лабораторной работы:

- повторение основными понятиями лексического и синтаксического анализа контекстно-свободных языков;
- закрепление практических навыков разработки и реализации лексических и синтаксических анализаторов;
- реализация учебного синтаксического анализатора для языка формальных троек Хоара и (типизированных) формул.

Что входит в лабораторную работу:

1. Формальное определение контекстно-свободного синтаксиса ВАШЕГО диалекта языка формальных троек Хоара для языка НеМо в нотации Бэкуса-Наура или в форме синтаксических диаграмм.
2. Программа лексико-синтаксического анализа, которая строит дерево грамматического разбора для программ ВАШЕГО диалекта формальных троек Хоара, и отвергает любой текст, который не является такой.

Этапы:

1. Разработка и формализация (в нотации Бэкуса-Наура или синтаксических диаграмм) синтаксиса ВАШЕГО диалекта языка формальных троек Хоара.
2. Разработка алгоритма лексического и синтаксического анализа ВАШЕГО диалекта языка формальных троек Хоара.
3. Дизайн и реализация лексического и синтаксического анализатора ВАШЕГО диалекта языка формальных троек Хоара.

Требования к диалекту формальных троек Хоара.

1. Полная синтаксическая совместимость по типам данных, программам, программным и свободным переменным с принятым ВАМИ диалектом НеМо.
2. Поддержка многоместных связок «AND» и «OR» для конъюнкции и дизъюнкции в предусловии и постусловии.
3. Поддержка двуместных связок «=>» и «<=>» для импликации и эквивалентности в предусловии и постусловии.

4. Поддержка одноместной связки «NOT» в качестве отрицания в предусловии и постусловии.
5. Поддержка кванторов «ALL» и «EXI» в качестве кванторов всеобщности и существования в предусловии и постусловии.
6. Поддержка термов любой синтаксической сложности в предусловии и постусловии.
7. Поддержка не менее чем 26 связанных переменных, соответствующих заглавным буквам латинского алфавита в предусловии и постусловии.
8. Выделение пробелами и/или переводами строки связок и кванторов («AND», «OR», «NOT», «=>», «<=>», «ALL», «EXI»).
9. Выделение пробелами и/или переводами строки отдельных описаний и отделение описаний от тела программы посредством перевода строки.

Требования к представлению алгоритма лексико-синтаксического анализа.

1. Описание алгоритма может быть чисто словесным, в форме хорошо структурированного текста, в виде псевдокода и/или блок-схем, или выполнено с использованием всех этих средств (объемом до пяти страниц).
2. Алгоритм и все его процедуры должен быть специфицирован условиями тотальной корректности по примеру алгоритма лексического и грамматического анализа языка HeMo из лекции 9.
3. Условия тотальной корректности алгоритма и всех его процедур должен быть снабжены доказательствами (общим объемом до пяти страниц) по примеру доказательств алгоритма лексического и грамматического анализа языка HeMo из лекции 10.
4. Разрешается не описывать, не специфицировать и не доказывать те процедуры, которые были описаны, специфицированы и доказаны при выполнении лабораторной работы 2.

Требования к программе лексико-синтаксического анализа.

1. Предусловие: Входные данные – ASCII файл с расширением TRI.
2. Постусловия:
 - 2.1. Если входной файл не является формальной тройкой Хоара на ВА-ШЕМ диалекте языка троек, то формируется LOG-файл с соответствующим сообщением (желательно, с указанием что именно не соответствует синтаксису).

- 2.2. Если входной файл является программой на принятом диалекте формальных троек Хоара, то формируется «таблицы» свободных, программных и связанных переменных тройки и три «дерева» грамматического разбора: формулы из предусловия, тела программы, формулы из постусловия.
- 2.2.1. Все три «таблицы» переменных представляют собой базу данных, которая для каждой описанной переменной устанавливает её тип, «внешнее» имя (определённое в программе), и «внутреннее» имя (используемое вместо внешнего), причём, как «внешнее» так и «внутреннее» имя являются ключами «таблицы».
- 2.2.2. «Дерево» грамматического разбора тела программы представляет собой базу данных, в которой вместо «внешних» имён переменных используются их «внутренние» имена, и которая по каждой подстроке, которая сама по себе является телом, предоставляет следующую информацию в соответствии с требованиями пункта 2.2.2. постусловия лабораторной работы 1 из лекции 10.
- 2.2.3. «Дерева» грамматического разбора формул из предусловия и постусловия представляют собой базу данных, в которой вместо «внешних» имён переменных используются их «внутренние» имена, и которая по каждой подстроке, которая сама по себе является телом, предоставляет следующую информацию:
- 2.2.3.1) если подстрока – это элементарная формула, то информация включает признак (флаг) используемого отношения и «ссылки» на термы из левой и правой части;
- 2.2.3.2) если подстрока – это конъюнкция («AND») или дизъюнкция («OR») нескольких подформул, то информация включает признак (флаг) конъюнкции/дизъюнкции соответственно и «ссылки» на все подподформулы, упорядоченные в соответствии с их порядком в подстроке слева направо;
- 2.2.3.3) если подстрока – это импликация («=>») или эквивалентность («<=>») двух подформул, то информация включает признак (флаг) импликации/эквивалентности и «ссылки» на обе подподформулы, упорядоченные в соответствии с их порядком в подстроке слева направо;

- 2.2.3.4) если подстрока – это отрицание («NOT»), то информация включает признак (флаг) отрицания и «ссылку» на подформулу этого отрицания.
- 2.2.3.5) если подстрока – это универсально («ALL») или экзистенциально («EXI») квантифицированная формула, то информация включает признак (флаг) универсального/экзистенциального квантора, «ссылку» на имя квантифицируемой связанной переменной и «ссылку» на квантифицируемую подформулу.

Комментарий и рекомендации.

Во-первых, еще раз прочитать рекомендации к лабораторной работе 2, так как они применимы и к данной лабораторной работе. Во-вторых, стоит проектировать и программировать алгоритм лексического и синтаксического анализа формальных троек Хоара, используя уже готовый алгоритм и программу лексического и синтаксического анализа НеМо-программ из лабораторной работы 2. Правда, в той лабораторной работе нет требования поддерживать использование свободных переменных. Но если соответствующий алгоритм был хорошо спроектирован и грамотно реализован, то расширить функциональность программы лексического и синтаксического анализа НеМо-программ на свободные переменные будет нетрудно.

Лекция 17. Семантика аксиоматической семантики вычислительных программ (на примере языка НеМо)

Ненадежность неформальных рассуждений

Сейчас самое время показать, почему неформальных рассуждений о надежности системы AS недостаточно для использования ее для верификации условий частичной корректности, почему все-таки необходима точная семантика формальных троек Хоара и формально доказанное утверждение о надежности системы AS.

Дело в том, что в неформальных рассуждениях о надежности системы AS использовалось без точного определения отсутствия какого-либо «побочного эффекта» в присваиваниях и тестах, отсутствия операторов переходов, нарушающих «блочную структуру» программы. Однако, если побочный эффект или нарушение структурированности возможно, то тогда правила и аксиомы системы AS могут перестать быть надежными.

Ниже будут приведены на неформальном уровне соответствующие контрпримеры⁷² с использованием плохо формализованного (но очень популярного) языка программирования C⁷³, когда из истинности посылок не следует истинность заключений. А сейчас следует подчеркнуть, что правильный путь избежать подобных коллизий – формализация семантики троек (чем мы займемся в следующей лекции).

Начнем с «ненадежности» аксиом системы AS.

В языке C есть одноместная операция «++» на целых числа. Она может использоваться как в префиксной так и в суффиксной форме. Смысл этой операции – увеличение на 1 аргумента с сохранением измененного значения в качестве нового значения. Разница операции «++» в префиксной и суффиксной форме состоит в том, что

- в префиксной форме сначала определяется значение аргумента операции, потом увеличивается на 1 и сохраняется, и только потом используется новое увеличенное значение;
- в суффиксной форме сначала определяется значение аргумента операции, потом используется, и только потом увеличивается на 1 и сохраняется.

Например, при выполнении присваивания⁷⁴ « $x=++y + 2$ » в состоянии, где значения x и y есть 3, результатом будет состояние, в котором значение x будет 6, а значение y будет 4, т.к. сначала будет извлечено значение 3 переменной y , потом оно будет увеличено на 1 (т.е. станет 4) и будет сохранено в качестве нового значения переменной y , потом к этому значению будет прибавлена 2 (в результате получится 6), и это значение станет новым значением переменной x . Наоборот, при выполнении присваивания « $x=y++ + 2$ » в этом же состоянии, результатом будет состояние, в котором значение x будет 5, а значение y опять будет 4, так как сначала будет извлечено значение 3 переменной y для дальнейшего использования, потом 3 (то есть значение переменной y) будет увеличено на 1 (то есть станет 4) и будет со-

⁷² В этих примерах мы специально используем нотацию языка C, что бы не путать их с HeMo.

⁷³ Существует огромное множество учебников по языку C, в частности, можно рекомендовать курс Костюкова Н.И., Калинина Н.А. Язык Си и особенности работы с ним. М.: Интуит, 2006. А вот что касается формальной семантики языка C, то и операционная, и аксиоматическая семантика этого языка – это актуальная исследовательская тема. Дело в том, что язык C очень платформо-зависимый, то есть очень многое в нем допускает в общем случае неоднозначную трактовку, и решается при реализации этого языка на конкретной ЭВМ для конкретной операционной системы разработчиками транслятора. Примером может служить ссылочный тип, значения которого – «адреса» в памяти машины.

⁷⁴ В нотации языка C для присваивания используется знак « $=$ ».

хранено в качестве нового значения переменной y , потом к 3 (то есть сохраненному для использования значению y) будет прибавлена 2 (в результате получится 5), и это значение станет новым значением переменной x . Аналогично, при выполнении проверки⁷⁵ « $x = ++y$ » в том же состоянии, что и выше, результатом будет новое состояние, в котором значения переменных x и y будут соответственно 3 и 4, а результат проверки – ЛОЖЬ; наоборот, при выполнении проверки « $x = .y++$ » в этом же состоянии результатом будет новое состояние, в котором значения переменных x и y также будут соответственно 3 и 4, но результат проверки – ИСТИНА.

В силу сказанного о смысле операции « $++$ » и ее побочном эффекте следующие «примеры» аксиом А и Т демонстрируют их «ненадежность» вне четких формальных рамок:

- { $(++y + 2) = 6 \wedge y = 4$ } $x = ++y + 2$ { $x = 6 \wedge y = 4$ } является аксиомой А, но не является правильным утверждением о частичной корректности;
- { $x = y++ \rightarrow (x = 3 \wedge y = 2)$ } $x = y++$? { $x = 3 \wedge y = 2$ } является аксиомой Т, но не является правильным утверждением о частичной корректности.

Теперь обсудим «ненадежность» правил вывода системы AS на примере одного правила SC. Для правил NC и LI можно построить аналогичные контрпримеры (которые предлагается сделать самостоятельно). А для правила PS/CW контрпример можно построить с использованием операции « $++$ » на подобие того, как это сделано выше для аксиом системы AS (что так же предлагается для самостоятельной работы).

В языке С разрешено использовать метки и операторы перехода «go to», причем разрешено как выходить из «блока» по go to, так и входить вовнутрь по метке нарушая структурированность. Так, например, следующая конструкция⁷⁶ записана на языке С (предполагается, что все переменные какого-либо одного целого типа⁷⁷):

if ($x = 2 * (x/2)$) { go to m; } ; if ($!(x = 2 * (x/2))$) { m: $x = x + 1$; } ;

Смысл этой конструкции можно описать так. Первый оператор проверяет значения переменной x на четность: если значение четное, то значение

⁷⁵ В нотации языка С для равенства используется знак « $==$ ».

⁷⁶ В нотации языка С знак « $;$ » является концом любого оператора, последовательная композиция является порядком выполнения операторов по умолчанию, «if (φ) {α};» используется вместо программы «IF φ THEN α ELSE TRUE?», а знак «!» используется для отрицания вместо « \neg ».

⁷⁷ В языке С несколько различных вариантов целого типа, представляющих разный диапазон математических целых чисел.

переменной x не изменяется, а управление передается на метку m , иначе – следующему оператору. Второй оператор проверяет значения переменной x на нечетность: если значение нечетное, то значение переменной x увеличивается на единицу, после чего выполнение приведенного фрагмента завершается. Но заметим, что в силу передачи управления на метку m внутри второго оператора, значение x в любом случае будет увеличено на 1, то есть следующее утверждение о частичной корректности

$$\begin{aligned} & \{ x = 2 \} \\ & \text{if } (x = 2 * (x/2)) \{ \text{go to } m; \} ; \text{ if } (! (x = 2 * (x/2))) \{ m: x = x + 1; \} ; \\ & \{ x = 3 \} \end{aligned}$$

является верным, а другое утверждение о частичной корректности

$$\begin{aligned} & \{ x = 2 \} \\ & \text{if } (x = 2 * (x/2)) \{ \text{go to } m; \} ; \text{ if } (! (x = 2 * (x/2))) \{ m: x = x + 1; \} ; \\ & \{ x = 2 \} \end{aligned}$$

является ложным. Но в тоже время верными являются два следующих утверждения частичной корректности:

- $\{ x = 2 \} \text{ if } (x = 2 * (x/2)) \{ \text{go to } m; \} ; \{ x = 2 \}$
- $\{ x = 2 \} \text{ if } (! (x = 2 * (x/2))) \{ m: x = x + 1; \} ; \{ x = 2 \}$

Следовательно, по правилу SC, утверждение частичной корректности

$$\begin{aligned} & \{ x = 2 \} \\ & \text{if } (x = 2 * (x/2)) \{ \text{go to } m; \} ; \text{ if } (! (x = 2 * (x/2))) \{ m: x = x + 1; \} ; \\ & \{ x = 2 \} \end{aligned}$$

тоже должно быть истинным, что противоречит сказанному выше об этом утверждении!

Повторим еще раз: все, что сказано в данном разделе о «ненадежности» аксиоматической семантики AS языка НеМо говорит только лишь о зыбкости неформального обоснования ее надежности. Как показали примеры на языке C, система AS действительно не подходит для всех языков программирования, но для тех языков, для которых она надежна, эта надежность должна быть вполне формально определена и обоснована.

Семантика формальных троек Хоара

Так как аксиоматическая семантика AS вычислительных программ языка НеМо определена в терминах формальных троек Хоара, (типизированных) формул и НеМо-программ, то нам необходимо дать точное определение сначала семантики (типизированных) формул и программ, потом – формальных троек Хоара, и только потом – самой аксиоматической семантики AS. По счастью у нас уже многое сделано: формальное определение трех эквивалентных (в модели наследственно-конечных таблиц) семантик вы-

числительных НеМо-программ дано в лекции 13, математическое определение семантики «неявно типизированных» формул логики высшего порядка в моделях дано в лекции 3, неформальное понятие частичной корректности было дано в лекции 1. Формальное определение семантики вычислительных программ практически готово для употребления в новом контексте, нам только понадобится внести небольшие коррективы в определение понятия состояния. Математическое определение семантики «неявно типизированных» формул логики высшего порядка надо будет сделать типизацию явной (но при этом мы будем следовать основной канве математического определения), т.е. «привязать» к конкретной модели, в которой происходят вычисления программ (состоящей из наследственно-конечных таблиц). После этого мы сможем кратко и формально определить истинность троек для языка НеМо, и, наконец, доказать утверждение о надежности аксиоматической семантики AS.

Для определенности (также как в лекции 13 при определении формальной семантики программ) выберем произвольно и зафиксируем до конца этой первой части лекции совокупность описаний свободных ρ , программных δ и связанных τ переменных так, что в $\rho\delta\tau$ каждая из описанных переменная имеет единственное описание. Мы также будем предполагать, что эта фиксированная совокупность описывает все переменные, которые будут встречаться вплоть до конца этой лекции.

В лекции 13 уже было определено понятие состояния и пространства состояний для произвольной совокупности непротиворечивой описаний. В частности, для любого подмножества⁷⁸ τ' описаний из τ пространство состояний $SP(\rho\delta\tau')$ – это совокупность всех отображений, которые сопоставляют каждой из переменной описанной в $\rho\delta\tau'$ переменной некоторое значение в соответствии с операционной семантикой её типа (то есть некоторую конечную таблицу). Как и раньше, каждое состояние s позволяет приписать значение $s(t)$ каждому синтаксически правильному выражению t языка НеМо, построенному только из переменных описанных в $\rho\delta\tau'$, также как это уже было определено в предыдущей лекциях 12 и 13.

Определение. Пусть $\{ \rho \in \Phi \} \delta \alpha \{ \psi \}$ – произвольная формальная тройка Хоара. Тогда семантика программы $\delta \alpha$ – это бинарное отношение вход-выход $\langle \alpha \rangle$ на пространстве состояний $SP(\rho\delta)$, определенное в соответствии с лекцией 13 (в терминах ТОС, СОС или ДС).

⁷⁸ возможно пустого или совпадающего с τ , но в общем случае – произвольного подмножества τ

Теперь перейдем к определению семантики формул в формальных тройках Хоара. Приведенное ниже определение основано на определении семантики формул из лекции 3, но отличается от него тем, что теперь состояния задают значения не всех переменных, а только тех, которые уже используются без квантора⁷⁹.

Определение. Для любой формулы ξ , в которой используются только переменные, описанные в $\text{r}\delta\tau$, пусть освобожденные в ξ переменные, это те переменные, которые встречаются в ξ , описаны среди связанных переменных τ , но не связаны в ξ ни каким квантором. Пусть τ^ξ – это совокупность описания из τ всех освобожденных в ξ переменных.

Проиллюстрируем понятие освобожденной переменной на примере 3 из предыдущей лекции 16. Совокупность ρ в данном случае состоит из следующих описаний свободных переменных

VAR m : INT, VAR r : (INT ARRAY OF INT),

VAR sigma : ((INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT)),

совокупность δ – из следующих описаний программных переменных

VAR k : INT, VAR t : INT,

а совокупность τ – из следующих описаний связанных переменных

VAR f : (INT ARRAY OF INT), VAR n : INT.

Формула в предусловии в примере 3 следующая:

$m \geq 0 \wedge$

$\forall \text{VAR } f : (\text{INT ARRAY OF INT})$

$(\text{APP}(\text{APP}(\text{sigma}, f), 0) = 0 \wedge$

$\forall \text{VAR } n : \text{INT } (n > 0 \rightarrow \text{APP}(\text{APP}(\text{sigma}, f), n) =$

$\text{APP}(\text{APP}(\text{sigma}, f), n-1) + \text{APP}(f, n))) \}$

Если принять в качестве ξ формулу

$\text{APP}(\text{APP}(\text{sigma}, f), 0) = 0 \wedge$

$\forall \text{VAR } n : \text{INT } (n > 0 \rightarrow \text{APP}(\text{APP}(\text{sigma}, f), n) =$

$\text{APP}(\text{APP}(\text{sigma}, f), n-1) + \text{APP}(f, n)))$

то единственная освобожденная переменная в ξ – это переменная sigma, а совокупность τ^ξ тогда состоит из единственного описания

VAR f : (INT ARRAY OF INT), VAR n : INT.

А если принять в качестве ξ формулу

$n > 0 \rightarrow \text{APP}(\text{APP}(\text{sigma}, f), n) = \text{APP}(\text{APP}(\text{sigma}, f), n-1) + \text{APP}(f, n)$

⁷⁹ Здесь так и хочется сказать «свободные переменные» про эти переменные, но это будет неправильно, так как при определении формальных троек Хоара в лекции 16 термин «свободные переменные» был закреплен за переменными, описанными в предусловии.

то освобожденных в ξ переменных будет две – sigma и n, а совокупность τ^ξ тогда включает два описания

VAR f : (INT ARRAY OF INT), VAR n : INT.

Определение. Пусть $\{ \rho \ \phi \} \delta \alpha \{ \psi \}$ - произвольная формальная тройка Хоара. Семантика любой формулы θ , которая является подформулой ϕ или ψ (включая сами эти формулы), определяется в терминах отношения выполнимости \models на состояниях из $SP(\rho \delta \tau^0)$. Пусть $s \in SP(\rho \delta \tau^0)$ – произвольное состояние. Тогда в зависимости от структуры θ имеем следующее:

- $s \models \text{TRUE}$, но неверно $s \models \text{FALSE}$;
- $s \models t_1 = t_2 \Leftrightarrow s(t_1)$ равно $s(t_2)$;
- $s \models t_1 < t_2 \Leftrightarrow s(t_1)$ меньше $s(t_2)$;
- $s \models t_1 > t_2 \Leftrightarrow s(t_1)$ больше $s(t_2)$;
- $s \models t_1 \leq t_2 \Leftrightarrow s(t_1)$ неменьше $s(t_2)$;
- $s \models t_1 \geq t_2 \Leftrightarrow s(t_1)$ не больше $s(t_2)$;
- $s \models (\neg(\psi)) \Leftrightarrow$ неверно $s \models \psi$;
- $s \models (\psi \wedge \xi) \Leftrightarrow s_\psi \models \psi$ и $s_\xi \models \xi$, $s \models (\psi \vee \xi) \Leftrightarrow s_\psi \models \psi$ или $s_\xi \models \xi$, где состояния $s_\psi \in SP(\rho \delta \tau^\psi)$ и $s_\xi \in SP(\rho \delta \tau^\xi)$ получаются из s в результате сужения отображения s на переменные, описанные в $\rho \delta \tau^\psi$ и $\rho \delta \tau^\xi$ соответственно;
- $s \models (\forall x \ \psi) \Leftrightarrow s' \models \psi$ для любого $s' \in SP(\rho \delta \tau^\psi)$, которое совпадает с s на всех переменных кроме x ;
- $s \models (\exists x \ \psi) \Leftrightarrow s' \models \psi$ для некоторого $s' \in SP(\rho \delta \tau^\psi)$, которое совпадает с s на всех переменных кроме x .

Определим множество истинности $[\theta]$ формулы θ так: $[\theta] = \{ s \in SP(\rho \delta \tau^0) : s \models \theta \}$.

Теперь все готово, что бы определить семантику формальных троек Хоара.

Определение. Пусть $\{ \rho \ \phi \} \delta \alpha \{ \psi \}$ - произвольная формальная тройка Хоара. Будем говорить что эта тройка истинна, если для любых состояний $s, t \in SP(\rho \delta)$ как только имеет место $s \models \phi$ и $s \langle \alpha \rangle t$, то имеет место и $t \models \psi$. Символически истинность тройки записывается $\models \{ \rho \ \phi \} \delta \alpha \{ \psi \}$.

Прежде чем разобрать примеры, обсудим как это формальное определение соотносится с общим понятием частичной корректности, введенном в лекции 1. Напомним, что программа частично корректна по отношению к предусловию ϕ и постусловию ψ , если на любых входных данных, которые удовлетворяют свойству ϕ , программа α или не останавливается (зацикливается, зависает и т.п.), или останавливается с выходными данными, которые удовлетворяют свойству ψ .

Входные данные в определении истинности формальной тройки Хоара представлены произвольным *начальным состоянием* s , выходные данные – произвольным *заключительным состоянием* t . То, что входные данные удовлетворяют условию ϕ , представлено в определении через *отношение выполнимости* $s \models \phi$. Аналогично, *отношение выполнимости* $t \models \psi$ представляет, что выходные данные удовлетворяют свойству ψ . То, что программа на входных данных s останавливается с выходными данными t , представлено в определении посредством *отношения вход-выход* $s\langle\alpha\rangle t$. Заметим, что в силу утверждения 2 из лекции 14 о корректности алгоритма трансляции НеМо-программ в программы виртуальной НеМо-машины, мы можем трактовать отношение $s\langle\alpha\rangle t$ именно так, что программа $(\delta\alpha)$ имеет вычисление, которое на входных данных s останавливается с выходными данными t . Это вычисление соответствует некоторой конечной полной трассе «реализации» программы на виртуальной машине⁸⁰.

Поэтому формальное определение истинности формальной тройки Хоара можно неформально прочесть так: тройка $\{ \rho \ \phi \} \delta \alpha \{ \psi \}$ истинна, если на любом начальном состоянии s , в котором выполнена формула ϕ , всякое полное вычисление α , которое завершается, завершается в каком-либо состоянии t , в котором выполнена формула ψ . Такое прочтение формального определения истинности формальных троек Хоара соответствует понятию истинности утверждения частичной корректности.

Позитивные примеры (истинные тройки)

Вернемся к рассмотрению примеров формальных троек Хоара из предыдущей лекции 16. Покажем, что все три примера – истинные тройки.

Пример 1 (перестановка целочисленных значений переменных местами).

В этом случае совокупность ρ состоит из двух описаний свободных переменных

VAR a : INT, VAR b : INT,

совокупность δ состоит из двух описаний программных переменных

VAR x : INT, VAR y : INT,

а совокупность τ связанных переменных пуста.

Нам надо проверить истинность тройки

$$\{ x=a \wedge y=b \} ((x := x+y ; y := x-y) ; x := x-y) \{ x=b \wedge y=a \}$$

⁸⁰ При этом могут существовать и аварийные, и бесконечные трассы, начинающиеся с тех же входных данных.

Денотационная семантика программы $(x := x+y ; y := x-y) ; x := x-y$ в пространстве $SP(\rho\delta)$ вычисляется достаточно легко. Достаточно заметить, что значения свободных переменных не изменяются никакой HeMo-программой, и поэтому состояния из $SP(\rho\delta)$ при вычислении денотационной семантики нашей программы можно представлять парами

(значение переменной x , значение переменной y).

Тогда имеем:

$$\begin{aligned} [(x := x+y ; y := x-y) ; x := x-y] &= ([x := x+y] \circ [y := x-y]) \circ [x := x-y] = \\ &= (\{ ((s(x), s(y)), (s(x)+s(y), s(y))) : s \in SP(\rho\delta) \} \circ \\ &\quad \circ \{ ((u(x), u(y)), (u(x), u(x)-u(y))) : u \in SP(\rho\delta) \}) \circ \\ &\quad \circ \{ ((w(x), w(y)), (w(x)-w(y), w(y))) : w \in SP(\rho\delta) \}) = \\ &= \{ ((s(x), s(y)), (s(x)+s(y), s(x)+s(y)-s(y))) : s \in SP(\rho\delta) \} \circ \\ &\quad \circ \{ ((w(x), w(y)), (w(x)-w(y), w(y))) : w \in SP(\rho\delta) \} = \\ &= \{ ((s(x), s(y)), (s(x)+s(y), s(x))) : s \in SP(\rho\delta) \} \circ \\ &\quad \circ \{ ((w(x), w(y)), (w(x)-w(y), w(y))) : w \in SP(\rho\delta) \} = \\ &= \{ ((s(x), s(y)), (s(x)+s(y)-s(x), s(x))) : s \in SP(\rho\delta) \} = \\ &= \{ ((s(x), s(y)), (s(y), s(x))) : s \in SP(\rho\delta) \}. \end{aligned}$$

Значит, для любых состояний $s, t \in SP(\rho\delta)$ имеем:

$$s \langle (x := x+y ; y := x-y) ; x := x-y \rangle t$$

тогда и только тогда, когда

$$s(x)=t(y) \text{ и } s(y)=t(x), \text{ а } s(a)=t(a) \text{ и } s(b)=t(b).$$

Если $s \models (x=a \wedge y=b)$, то $s(x)=s(a)$ и $s(y)=s(b)$. Поэтому $t(x)=s(b)$ и $t(y)=s(a)$, т.е. $t \models (x=b \wedge y=a)$. Следовательно,

$$\{ x=a \wedge y=b \} ((x := x+y ; y := x-y) ; x := x-y) \{ x=b \wedge y=a \}$$

истинность тройки установлена.

Пример 2: наибольший общий делитель двух положительных целых чисел a и b .

В этом случае совокупность ρ состоит из двух описаний свободных переменных

$$\text{VAR } a : \text{INT}, \text{VAR } b : \text{INT},$$

совокупность δ состоит из двух описаний программных переменных

$$\text{VAR } x : \text{INT}, \text{VAR } y : \text{INT},$$

а совокупность τ состоит из описаний пяти (!) связанных переменных

$$\text{VAR } m : \text{INT}, \text{VAR } n : \text{INT}, \text{VAR } z : \text{INT}, \text{VAR } p : \text{INT}, \text{VAR } q : \text{INT}.$$

Нам надо проверить истинность тройки

$$\{ a>0 \wedge b>0 \}$$

$$x := a ; y := b ;$$

$$((x>y ? ; x := x-y) \cup (x<y ? ; y := y-x))^* ;$$

$x=y ?$

$\{ (\exists \text{ VAR } m : \text{INT } (m>0 \wedge a=m \times x)) \wedge (\exists \text{ VAR } n : \text{INT } (n>0 \wedge b=n \times x)) \wedge$
 $\forall \text{ VAR } z : \text{INT}$

$((\exists \text{ VAR } p : \text{INT } (p>0 \wedge a=p \times z)) \wedge (\exists \text{ VAR } q : \text{INT } (q>0 \wedge b=q \times z)) \rightarrow z \leq x) \}$

Денотационная семантика программы в этом примере не столь проста как в примере 1, но нам достаточно вычислить бинарное отношение, которое «покрывает» семантику данной программы. Как и в первом примере, состояния из $\text{SP}(\rho\delta)$ опять можно представлять парами значений переменных x и y . Кроме того, введем функцию⁸¹ $\text{НОД} : \text{INT} \times \text{INT} \rightarrow \text{INT}$, которая по паре целых положительных чисел возвращает их наибольший общий делитель.

Во-первых, имеем

$$\begin{aligned} [x := a ; y := b ; ((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x))^* ; x = y ?] = \\ = [x := a ; y := b] \circ [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^* \circ [x = y ?]. \end{aligned}$$

Во-вторых

$$[x := a ; y := b] = \{ ((s(x), s(y)), (s(a), s(b))) : s \in \text{SP}(\rho\delta) \}$$

и

$$[x = y ?] = \{ ((t(x), t(y)), (t(x), t(y))) : t \in \text{SP}(\rho\delta), t(x) = t(y) \}.$$

Далее индукцией по $i \geq 0$ доказывается что

$$\begin{aligned} [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^i \subseteq \\ \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in \text{SP}(\rho\delta), \\ (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\ \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \}. \end{aligned}$$

Действительно для $i=0$ имеем:

$$\begin{aligned} [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^0 = \\ = \{ ((u(x), u(y)), (u(x), u(y))) : u \in \text{SP}(\rho\delta) \} \subseteq \\ \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in \text{SP}(\rho\delta), \\ (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\ \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \}. \end{aligned}$$

Если для какого-либо $i \geq 0$ имеет место

$$\begin{aligned} [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^i \subseteq \\ \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in \text{SP}(\rho\delta), \\ (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\ \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \}, \end{aligned}$$

⁸¹ Эта функция вводится не в язык троек, а только на метауровне для выполнения семантических рассуждений. Поэтому для нее выбрано хорошо всем знакомая русская аббревиатура.

то тогда используется свойство монотонности композиции бинарных отношений $R \subseteq S \Rightarrow (R^\circ T) \subseteq (S^\circ T)$ следующим образом:

$$\begin{aligned}
& [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^{i+1} = \\
& = [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^i \circ \\
& \quad \circ [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)] = \\
& = [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^i \circ \\
& \quad \circ (\{ ((w(x), w(y)), ((w(x) - w(y)), w(y))) : w \in SP(\rho\delta), w(x) > w(y) \} \cup \\
& \quad \cup \{ ((w(x), w(y)), (w(x), (w(y) - w(x)))) : w \in SP(\rho\delta), w(x) < w(y) \}) \subseteq \\
& \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in SP(\rho\delta), \\
& \quad (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\
& \quad \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \} \circ \\
& \quad \circ (\{ ((w(x), w(y)), ((w(x) - w(y)), w(y))) : w \in SP(\rho\delta), w(x) > w(y) \} \cup \\
& \quad \cup \{ ((w(x), w(y)), (w(x), (w(y) - w(x)))) : w \in SP(\rho\delta), w(x) < w(y) \}) = \\
& = \{ ((u(x), u(y)), ((w(x) - w(y)), w(y))) : u, w \in SP(\rho\delta), w(x) > w(y), \\
& \quad (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \} \cup \\
& \cup \{ ((u(x), u(y)), (w(x), (w(y) - w(x)))) : u, w \in SP(\rho\delta), w(x) < w(y), \\
& \quad (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \} \subseteq \\
& \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in SP(\rho\delta), \\
& \quad (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\
& \quad \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \}
\end{aligned}$$

так как для любых положительных целых чисел g и h имеет место

$$\text{НОД}(g, h) = \text{НОД}(g - h, h) = \text{НОД}(g, h - g).$$

Поэтому, в-третьих, имеем

$$\begin{aligned}
& [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^* = \\
& = \bigcup_{i \geq 0} [(x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)]^i \subseteq \\
& \subseteq \{ ((u(x), u(y)), (w(x), w(y))) : u, w \in SP(\rho\delta), \\
& \quad (u(x) > 0 \text{ и } u(y) > 0 \Rightarrow w(x) > 0 \text{ и } w(y) > 0), \\
& \quad \text{НОД}(u(x), u(y)) = \text{НОД}(w(x), w(y)) \}.
\end{aligned}$$

Собрав вместе все три результата (и еще раз используя свойство монотонности композиции бинарных отношений) получаем

$$\begin{aligned}
& [x := a ; y := b ; ((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x))^* ; x = y ?] \subseteq \\
& \subseteq \{ ((s(x), s(y)), (t(x), t(y))) : s, t \in SP(\rho\delta), s(x) = s(a), s(y) = s(b), \\
& \quad (s(x) > 0 \text{ и } s(y) > 0 \Rightarrow t(x) > 0 \text{ и } t(y) > 0), \\
& \quad \text{НОД}(s(x), s(y)) = \text{НОД}(t(x), t(y)), \\
& \quad t(x) = t(y) \}.
\end{aligned}$$

Значит, для любых состояний $s, t \in SP(\rho\delta)$ если

$$s \langle x := a ; y := b ; ((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x))^* ; x = y ? \rangle t$$

то

$$s(x)=s(a), s(y)=s(b), (s(x)>0 \text{ и } s(y)>0 \Rightarrow t(x)>0 \text{ и } t(y)>0), \\ \text{НОД}(s(x), s(y)) = \text{НОД}(t(x), t(y)), t(x)=t(y).$$

А т.к. для любых положительных целых чисел g и h имеет место $g=h \Rightarrow g=\text{НОД}(g,h)$,

то

$$s(x)=s(a), s(y)=s(b), (s(x)>0 \text{ и } s(y)>0 \Rightarrow t(x)>0 \text{ и } t(y)>0), \\ (t(x)>0 \text{ и } t(y)>0 \Rightarrow t(x)=\text{НОД}(s(x), s(y))).$$

Поэтому если $s \models (a>0 \wedge b>0)$, то $t(x)=\text{НОД}(s(a), s(b))$.

Проверим формально, что в таком состоянии $t \in \text{SP}(\rho\delta)$, где $t(x)=\text{НОД}(s(a), s(b))$, обязательно имеет место

$$t \models (\exists \text{VAR } m : \text{INT } (m>0 \wedge a=m \times x)) \wedge (\exists \text{VAR } n : \text{INT } (n>0 \wedge b=n \times x)) \wedge \\ \forall \text{VAR } z : \text{INT}$$

$$((\exists \text{VAR } p : \text{INT } (p>0 \wedge a=p \times z)) \wedge (\exists \text{VAR } q : \text{INT } (q>0 \wedge b=q \times z)) \rightarrow z \leq x).$$

Так как формула, которую надо проверить представляет собой конъюнкцию трех подформул, то (согласно определению семантики формул в тройках) необходимо проверить что

$$1) \quad t \models \exists \text{VAR } m : \text{INT } (m>0 \wedge a=m \times x) ,$$

$$2) \quad t \models \exists \text{VAR } n : \text{INT } (n>0 \wedge b=n \times x) ,$$

$$3) \quad t \models \forall \text{VAR } z : \text{INT}$$

$$4) \quad ((\exists \text{VAR } p : \text{INT } (p>0 \wedge a=p \times z)) \wedge (\exists \text{VAR } q : \text{INT } (q>0 \wedge b=q \times z)) \rightarrow z \leq x).$$

Пусть $\tau^m, \tau^n, \tau^z, \tau^p$, и τ^q – это описания связанных переменных m, n, z, p и q из этих формул.

Свойства 1 и 2 проверяются аналогично (с точностью до замены переменных «а» и «m» на «b» и «n» соответственно. Поэтому проверим только 1. Согласно определению семантики, необходимо «предъявить» такое состояние $t' \in \text{SP}(\rho\delta\tau^m)$, что

$$t'(x)=t(x), t'(y)=t(y), t'(a)=t(a), t'(b)=t(b) \text{ и } t' \models (m>0 \wedge a=m \times x).$$

В качестве такого состояния достаточно взять состояние t' что

$$t'(x)=t(x), t'(y)=t(y), t'(a)=t(a), t'(b)=t(b) \text{ и } t'(m)= \text{частное от деления } t(a) \text{ на } t(x).$$

Так как $t(x)=\text{НОД}(t(a), t(b))$, то $t'(m)$ определено и $t'(a)=t'(m) \times t'(x)$. Поэтому 1 верно.

Для проверки 3 согласно определению семантики надо показать что для любого состояний $t'' \in \text{SP}(\rho\delta\tau^z)$, в котором $t''(x)=t(x), t''(y)=t(y), t''(a)=t(a), t''(b)=t(b)$, имеет место

$t'' \models (\exists \text{VAR } p : \text{INT } (p > 0 \wedge a = p \times z)) \wedge (\exists \text{VAR } q : \text{INT } (q > 0 \wedge b = q \times z)) \rightarrow z \leq x$,
т.е. если $t'' \models \exists \text{VAR } p : \text{INT } (p > 0 \wedge a = p \times z)$ и $t'' \models \exists \text{VAR } q : \text{INT } (q > 0 \wedge b = q \times z)$, то $t'' \models z \leq x$.

Но если $t'' \models \exists \text{VAR } p : \text{INT } (p > 0 \wedge a = p \times z)$, то для некоторого $t''' \in \text{SP}(\rho\delta \tau^p)$ имеет место $t''' \models (p > 0 \wedge a = p \times z)$ и $t'''(x) = t''(x)$, $t'''(y) = t''(y)$, $t'''(a) = t''(a)$, $t'''(b) = t''(b)$, $t'''(z) = t''(z)$, т.е. $t'''(z)$ является делителем $t''(a)$. Вспоминая теперь связь $t(a) = t''(a) = t'''(a)$ и $t''(z) = t'''(z)$, а так же что $0 < t(a)$ и $0 < t'''(p)$ заключаем, что $t''(z)$ является положительным делителем $t(a)$.

Аналогично, если $t'' \models \exists \text{VAR } q : \text{INT } (q > 0 \wedge b = q \times z)$, то $t''(z)$ является положительным делителем $t(b)$.

Поэтому $t''(z)$ является положительным *общим* делителем $t(a)$ и $t(b)$. Следовательно, $t''(z)$ не больше чем $\text{НОД}(t(a), t(b))$. Так как $t(a) = t''(a)$ и $t(b) = t'(b)$, а $\text{НОД}(t(a), t(b)) = t(x) = t''(x)$, то $t''(z)$ не больше чем $t''(x)$, т.е. $t'' \models z \leq x$.

Таким образом мы показали, что имеют место все три свойства 1, 2 и 3, т.е. что из $s \models (a > 0 \wedge b > 0)$ и

$s \langle x := a ; y := b ; ((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)) * ; x = y ? \rangle t$ следует

$t \models (\exists \text{VAR } m : \text{INT } (m > 0 \wedge a = m \times x)) \wedge (\exists \text{VAR } n : \text{INT } (n > 0 \wedge b = n \times x)) \wedge \forall \text{VAR } z : \text{INT} \\ ((\exists \text{VAR } p : \text{INT } (p > 0 \wedge a = p \times z)) \wedge (\exists \text{VAR } q : \text{INT } (q > 0 \wedge b = q \times z))) \rightarrow z \leq x$.

В силу произвольного выбора пары состояний $s, t \in \text{SP}(\rho\delta)$ заключаем, что

$\models \{ a > 0 \wedge b > 0 \}$
 $x := a ; y := b ;$
 $((x > y ? ; x := x - y) \cup (x < y ? ; y := y - x)) * ;$
 $x = y ?$
 $\{ (\exists \text{VAR } m : \text{INT } (m > 0 \wedge a = m \times x)) \wedge (\exists \text{VAR } n : \text{INT } (n > 0 \wedge b = n \times x)) \wedge \forall \text{VAR } z : \text{INT} \\ ((\exists \text{VAR } p : \text{INT } (p > 0 \wedge a = p \times z)) \wedge (\exists \text{VAR } q : \text{INT } (q > 0 \wedge b = q \times z))) \rightarrow z \leq x \}$.

Позитивный vs. негативный пример

В этой части мы сначала формально (используя семантику) покажем истинность примера 3 из предыдущей лекции, а потом рассмотрим несколько его «неправильных» вариантов с измененной программой, предусловием и постусловием и покажем, что все эти варианты не являются истинными тройками.

Пример 3: сумма первых m элементов целочисленного массива g .

В этом случае совокупность ρ состоит из трех описаний свободных переменных

VAR m : INT , VAR r : (INT ARRAY OF INT) ,
 VAR σ : ((INT ARRAY OF INT) ARRAY OF (INT ARRAY OF INT)) ,
 совокупность δ состоит из двух описаний программных переменных

VAR k : INT , VAR t : INT ,

а совокупность τ состоит из описаний двух связанных переменных

VAR f : (INT ARRAY OF INT) , VAR n : INT .

Нам надо проверить истинность тройки

{ $m \geq 0 \wedge$
 \forall VAR f : (INT ARRAY OF INT)
 (APP(APP(σ , f),0)=0 \wedge
 \forall VAR n : INT ($n > 0 \rightarrow$ APP(APP(σ , f), n)=
 APP(APP(σ , f), $n-1$)+APP(f , n))) }

$k := 1$; $t := 0$;

($k \leq m$? ; $t := t + \text{APP}(r,k)$; $k := k + 1$) * ; $k > m$?

{ $t = \text{APP}(\text{APP}(\sigma, r), m)$ }

Напомним также, что в предыдущей лекции 17 формула

\forall VAR f : (INT ARRAY OF INT)
 (APP(APP(σ , f),0)=0 \wedge
 \forall VAR n : INT ($n > 0 \rightarrow$ APP(APP(σ , f), n)=
 APP(APP(σ , f), $n-1$)+APP(f , n)))

была обозначена ($A\Sigma$).

Денотационная семантика программы в этом примере не столь проста как в примере 1, но и не столь сложна как в примере 2 (хотя более рутинна). Как и в предыдущих примерах, при определении семантики программы состояния из $SP(\rho\delta)$ опять можно представлять значениями переменных k и t , которые могут менять значения во время выполнения программы. Кроме того, введем функцию⁸² Σ : $([INT \rightarrow INT] \times (INT \times INT)) \rightarrow INT$, которая по любой семье целых чисел индексированных целыми числами $\{a_i : i \in \text{ЦЕЛ}\}$ и целым числом m и n возвращает сумму $\Sigma_{i \in [m..n]} a_i = (a_m + \dots + a_n)$, если $m \leq n$, и $\Sigma_{i \in [m..n]} a_i = 0$ если $m > n$.

Имеем:

⁸² Так же как в случае с функцией НОД, функция Σ вводится не в язык троек, а только на метаязыке для выполнения семантических рассуждений. Поэтому для нее выбрано хорошо всем знакомое обозначение.

$$\begin{aligned}
& [k := 1 ; t := 0 ; (k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^* ; k > m ?] = \\
& = [k := 1 ; t := 0] \circ [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*] \circ [k > m ?] = \\
& = \{ (u, (1, 0)) : u \in \text{SP}(\rho\delta) \} \circ \\
& \quad \circ [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*] \circ \\
& \quad \quad \circ \{ (w, w) : w \in \text{SP}(\rho\delta) \text{ и } w(k) > w(m) \}.
\end{aligned}$$

Для вычисления семантики $(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*$ индукцией по $i \geq 0$ покажем, что

при $i=0$

- $[(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*]^i = \{ (u, u) : u \in \text{SP}(\rho\delta) \}$,
- а при $i > 0$
- $[(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*]^i = \{ (u, w) : u, w \in \text{SP}(\rho\delta), u \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, w(k) - 1 \leq u(m), w(k) = u(k) + i, \text{ и } w(t) = u(t) + \sum_{j \in [u(k)..w(k)-1]} \text{APP}(u(r), j) \}.$

База индукции при $i=0$ очевидна. Если принять в качестве предположения индукции, что для некоторого $i \geq 0$ эти равенства имеют место, то для $(i+1)$ имеем следующее.

При $i=0$

$$\begin{aligned}
& [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*]^{(i+1)} = [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)] = \\
& = \{ (u, w) : u, w \in \text{SP}(\rho\delta), u(k) \leq u(m), u \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, w(k) = u(k) + 1 \text{ и } w(t) = u(t) + \text{APP}(u(r), u(k)) \} = \\
& = \{ (u, w) : u, w \in \text{SP}(\rho\delta), u \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, u(k) \leq u(m), w(k) = u(k) + (i+1) \text{ и } w(t) = u(t) + \sum_{j \in [u(k)..w(k)-1]} \text{APP}(u(r), j) \}
\end{aligned}$$

При $i > 0$

$$\begin{aligned}
& [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*]^{(i+1)} = \\
& = [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*]^i \circ [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)] = \\
& = \{ (u, v) : u, v \in \text{SP}(\rho\delta), u \text{ и } v \text{ совпадают всюду кроме как на переменных } k \text{ и } t, v(k) - 1 \leq u(m), v(k) = u(k) + i \text{ и } v(t) = u(t) + \sum_{j \in [u(k)..v(k)-1]} \text{APP}(u(r), j) \} \circ \\
& \quad \circ \{ (v, w) : v, w \in \text{SP}(\rho\delta), v \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, v(k) \leq v(m), w(k) = v(k) + 1 \text{ и } w(t) = v(t) + \text{APP}(v(r), v(k)) \} = \\
& = \{ (u, w) : u, w \in \text{SP}(\rho\delta), u \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, w(k) - 1 \leq u(m), w(k) = u(k) + (i+1) \text{ и } w(t) = u(t) + \sum_{j \in [u(k)..w(k)-1]} \text{APP}(u(r), j) \}.
\end{aligned}$$

Поэтому имеем:

$$\begin{aligned}
& [(k \leq m ? ; t := t + \text{APP}(r, k) ; k := k + 1)^*] = \\
& = \{ (u, w) : u, w \in \text{SP}(\rho\delta), u \text{ и } w \text{ совпадают всюду кроме как на переменных } k \text{ и } t, \text{ и существует такое } i > 0 \text{ что } u(k) + i - 1 \leq u(m), w(k) = u(k) + i, a
\end{aligned}$$

$$w(t)=u(t)+\sum_{j \in [u(k)..w(k)-1]} APP(u(r),j) \} \cup \\ \cup \{ (u, u) : u \in SP(\rho\delta) \}.$$

Следовательно,

$$[k := 1 ; t := 0 ; (k \leq m ? ; t := t + APP(r,k) ; k := k + 1)^* ; k > m ?] = \\ = \{ (u, w) : u, w \in SP(\rho\delta), u \text{ и } w \text{ совпадают всюду кроме как на перемен-} \\ \text{ных } k \text{ и } t, w(k)=u(m)+1, \text{ а } w(t)=\sum_{j \in [1..u(k)]} APP(u(r),j) \}.$$

Или, словами, вычислительная семантика рассмотренной программы состоит из всех пар состояний (u, w) таких, что u и v совпадают всюду, кроме как на переменных k и t , «начальные» значения $u(k)$ и $u(t)$ произвольные, а «конечные» значения $w(k)$ и $w(t)$ зависят от значения $u(m)$:

- если $u(m) < 0$, то $w(k)=1$ и $w(t)=0$,
- иначе $w(k)=u(m)+1$ а $w(t)=\sum_{j \in [1..u(m)]} APP(u(r),j)$.

Пусть $u \langle k := 1 ; t := 0 ; (k \leq m ? ; t := t + APP(r,k) ; k := k + 1)^* ; k > m ? \rangle$ и пусть $u \models (m \geq 0 \wedge (A\Sigma))$. Тогда $u(m) \geq 0$ и $u \models (A\Sigma)$. Индукцией по $n \geq 0$ можно показать⁸³, что если $u \models (A\Sigma)$ то для любой функции $f: ЦЕЛ \rightarrow ЦЕЛ$ имеет место равенство $APP(APP(\sigma, f), n) = \sum_{j \in [1..n]} APP(f, j)$. Так как программа не меняет значение свободных переменных u , m и σ , то, следовательно, в состоянии w имеем $w(APP(APP(\sigma, r), m)) = APP(APP(w(\sigma), w(r)), w(m)) =$
 $= APP(APP(u(\sigma), u(r)), u(m)) = \sum_{j \in [1..u(m)]} APP(u(r), j)$.

Но в соответствии с семантикой программы имеем

$$\sum_{j \in [1..u(m)]} APP(u(r), j) = \sum_{j \in [1..w(k)-1]} APP(u(r), j) = w(t).$$

Следовательно, $w(APP(APP(\sigma, r), m)) = w(t)$, и значит $w \models (t = APP(APP(\sigma, r), m))$.

Таким образом, тройка из примера 3 из лекции 16 верна, что и требовалось доказать.

Теперь, когда «позитивный» пример разобран в деталях, посмотрим, что произойдет, если внести ошибки в

1. предусловие,
2. постусловие,
3. программу.

Например, пусть в первом случае мы «забыли» потребовать в предусловии $m \geq 0$. Тогда все свойства семантики программы и семантики функции σ , отмеченные выше в «позитивной» части, по-прежнему имеют место.

⁸³ Это предлагается проделать самостоятельно в качестве упражнения.

Пусть (u, w) – произвольная пара вход-выход из семантики нашей программы, и пусть в состоянии u выполнено предусловие, т.е. $u \models (A\Sigma)$. Условие $u \models (A\Sigma)$ допускает в качестве функции $u(\text{sigma})$ такую, что $\text{APP}(\text{APP}(u(\text{sigma}), f), n)$ – это какое-либо отрицательное число (например -10^6) для всех функций f : ЦЕЛ \rightarrow ЦЕЛ и всех целых $n < 0$. Но тогда в состоянии w в постусловие $t = \text{APP}(\text{APP}(\text{sigma}, r), m)$ неверно, т.к. $w(t) = 0$, а $w(\text{APP}(\text{APP}(\text{sigma}, r), m)) = -10^6$.

Во втором случае предположим, что ошибка «вкралась» в постусловие и вместо $t = \text{APP}(\text{APP}(\text{sigma}, r), m)$ мы написали $t = \text{APP}(\text{APP}(\text{sigma}, r), k)$ в постусловии. (Но предусловие при этом записано «правильно», т.е. так, как в «позитивном» примере.) Тогда опять все свойства семантики программы и семантики функции sigma , отмеченные выше в «позитивной» части, по-прежнему имеют место. Пусть (u, w) – произвольная пара вход-выход из семантики нашей программы, и пусть в состоянии u выполнено предусловие. Тогда согласно «позитивному» примеру по семантике программы мы знаем, что $w(k) = u(m) + 1$ а $w(t) = \sum_{j \in [1..u(m)]} \text{APP}(u(r), j)$. Если бы при этом было $w \models (t = \text{APP}(\text{APP}(\text{sigma}, r), k))$, то было бы $w(t) = \sum_{j \in [1..u(k)]} \text{APP}(u(r), j)$, и, следовательно, $\text{APP}(u(r), u(m) + 1) = 0$, т.е. что $(m + 1)$ -ый элемент массива r обязательно 0, что в общем случае неверно.

И наконец в третьем случае предположим, что при исходных «правильных» пред- и пост- условиях используется несколько другая программа⁸⁴:

$k := 1 ; t := 0 ; (t := t + \text{APP}(r, k) ; k := k + 1 ; k \leq m ?)^* ; k > m ?$

Вычислительная семантика такой «подправленной» программы состоит⁸⁵ из всех пар состояний (u, w) таких, что u и v совпадают всюду, кроме как на переменных k и t , «начальное» значение $u(m) \leq 0$, «начальные» значения $u(k)$ и $u(t)$ произвольные, а «конечные» значения $w(k) = 1$ и $w(t) = 0$. Таким образом, если в каком-либо состоянии выполнено предусловие $(m \geq 0 \wedge (A\Sigma))$, то в этом состоянии или $m = 0$, или $m > 0$. В первом из этих случаев программа останавливается правильно вычислив в переменной t сумму (равную 0) m первых элементов массива r , а во втором случае – вообще не останавливается. Поэтому следующее утверждение частичной корректности с «подправленной» программой верно:

$\{ m \geq 0 \wedge$
 $\forall \text{VAR } f : (\text{INT ARRAY OF INT})$
 $(\text{APP}(\text{APP}(\text{sigma}, f), 0) = 0 \wedge$

⁸⁴ Переставлен тест внутри недетерминированной итерации.

⁸⁵ Проверить правильность семантики предлагается самостоятельно в качестве упражнения.

$$\forall \text{VAR } n : \text{INT} (n > 0 \rightarrow \text{APP}(\text{APP}(\text{sigma}, f), n) = \\ \text{APP}(\text{APP}(\text{sigma}, f), n-1) + \text{APP}(f, n))) \}$$

$k := 1 ; t := 0 ;$
 $(t := t + \text{APP}(r, k) ; k := k + 1 ; k \leq m ?)^* ; k > m ?$
 $\{ t = \text{APP}(\text{APP}(\text{sigma}, r), m) \}$

В заключении лекции отметим, что разобранные примеры не только иллюстрируют семантику формальных троек Хоара для языка HeMo, но и демонстрируют, что проверка или доказательство истинности тройки еще не означает отсутствие ошибок ни в предусловии, ни в постусловии, ни в самой программе. (Особенно поучителен негативный пример 3 с «подправленной» программой.) Однако ничего удивительного в этом нет, ведь следующие тройки⁸⁶ всегда истинны:

- $\{\text{FALSE}\} \alpha \{\psi\}$,
- $\{\phi\} \alpha \{\text{TRUE}\}$,
- $\{\phi\} \text{LOOP} \{\psi\}$,

где ϕ и ψ - произвольные формулы, а LOOP – произвольная никогда не останавливающаяся программа.

Лекция 18. От надежности – к полноте аксиоматической семантики (на примере языка HeMo)

Надежность аксиоматической семантики языка HeMo

Напомним, что в лекции 4 было определено общее понятие надежности аксиоматической системы. Определение включало в себя отображение, которое «переводило» формулы – в заключения правил вывода аксиоматической системы. Если при этом оказывалось, что доказуемы только «переводы» истинных формул, то аксиоматическая система называлась надежной.

Аксиоматическая семантика формальных троек Хоара для языка HeMo является частным случаем аксиоматической системы. В качестве формул в данном случае выступают формальные тройки Хоара для языка HeMo. В таком случае никакого дополнительного «перевода» троек в заключения правил аксиоматической семантики не требуется, так как все заключения являются формальными тройками Хоара⁸⁷.

⁸⁶ Описания переменных мы опускаем как несущественную деталь в этих примерах.

⁸⁷ Точнее говоря, перевод является тождественной функцией, не меняющей тройку.

Имеет место следующее утверждение о надежности аксиоматической семантики языка НеМо.

Утверждение 16

Если в каком-либо дереве вывода в аксиоматической семантике AS для языка НеМо все листья – истинные типизированные формулы, или истинные формальные тройки Хоара для языка НеМо, или аксиомы системы AS, то и корень этого дерева вывода – истинная формальная тройка Хоара для языка НеМо.

Доказательство проведем индукцией по высоте дерева вывода. Для простоты мы предполагаем, что совокупности описаний свободных и программных переменных ρ и δ выбраны произвольно и зафиксированы, все программы и формулы совместимы с этими описаниями, а SP обозначает пространство состояний $SP(\rho\delta)$, которое соответствует описаниям $\rho\delta$.

База индукции: высота дерева равна 1. Но если дерево удовлетворяет посылке утверждения, то оно или состоит из истинной тройки, или представлено аксиомой системы AS. В первом из этих случаев доказывать нечего, а во втором надо рассмотреть два случая в зависимости от того, какая это из двух аксиом системы AS – аксиома для присваивания A, или аксиома для теста T. Разберем оба эти случая.

Аксиома для присваивания A: $\{\xi_{u/x}\}(x:=t)\{\xi\}$. Пусть u и w – произвольные состояния из SP такие, что $u \langle x:=t \rangle w$. Тогда⁸⁸ $w = UPD(u, x, u(t))$. Пусть $d=u(t)$ – значение выражения t в состоянии u (в виде наследственно-конечной таблицы). Имеем: $u \models \xi_{u/x} \Leftrightarrow^{89} u \models \xi_{d/x} \Leftrightarrow^{90} UPD(u, x, u(t)) \models \xi_{d/x} \Leftrightarrow^{91}$

⁸⁸ Здесь используется формальная семантика языка НеМо, в которой присваивание не имеет никакого побочного эффекта.

⁸⁹ Вообще говоря, эта эквивалентность в этой цепочке – отдельная лемма, которую можно сформулировать следующим образом. Для любой типизированной формулы χ , в которой все свободные переменные описаны в контексте $\rho\delta$, для любой переменной y , описанной в $\rho\delta$ (то есть не связанной никаким квантором), для любого выражения E , в котором используются только переменные описанные в $\rho\delta$, для любого состояния $v \in SP(\rho\delta)$, если $d=v(E)$ – значение выражения E в состоянии v , то имеет место эквивалентность: $v \models \chi_{E/y} \Leftrightarrow UPD(v, y, v) \models \chi_{d/y}$. Эту лемму следует доказывать индукцией по структуре формулы χ . Но за недостатком места для этого чисто технического доказательства ограничимся рекомендацией выполнить это доказательство в качестве упражнения самостоятельно.

⁹⁰ Вторая эквивалентность в цепочке использует тот факт, что формула $\xi_{d/x}$ не содержит вхождений переменной x , вместо которого подставлено значение d .

⁹¹ Используется та же лемма, что и в сноске 89: достаточно принять $v=UPD(u, x, u(t))$ и $E \equiv x$. Заметим сразу, что в этой же лекции эта лемма без явной ссылки будет использоваться еще два раза: в доказательстве пункта 2 Утверждение 17 и пункта 1 Утверждение 19.

$\text{UPD}(u, x, u(t)) \models \xi \Leftrightarrow w \models \xi$. В частности, из $u \models \xi_{t/x}$ следует $w \models \xi$. В силу произвольного выбора u и w заключаем истинность тройки $\{\xi_{t/x}\}(x:=t)\{\xi\}$.

Аксиома для теста T : $\{(\phi \rightarrow \psi)\}(\phi?)\{\psi\}$. Пусть u и w – произвольные состояния из SP такие, что $u \models \phi?$ w . Тогда⁹² $w = u$ и $u \models \phi$. Поэтому в состоянии u имеем: $u \models (\phi \rightarrow \psi) \Leftrightarrow u \models \psi$. В частности, из $u \models (\phi \rightarrow \psi)$ следует $w \models \psi$. В силу произвольного выбора u и w заключаем истинность тройки $\{(\phi \rightarrow \psi)\}(\phi?)\{\psi\}$.

Предположение индукции: предположим, что утверждение верно для всякого дерева вывода, удовлетворяющего условиям утверждения, высота которого не превосходит $h \geq 1$.

Шаг индукции: покажем, что утверждение верно для всякого дерева вывода, удовлетворяющего условиям утверждения, высота которого не превосходит $(h+1)$.

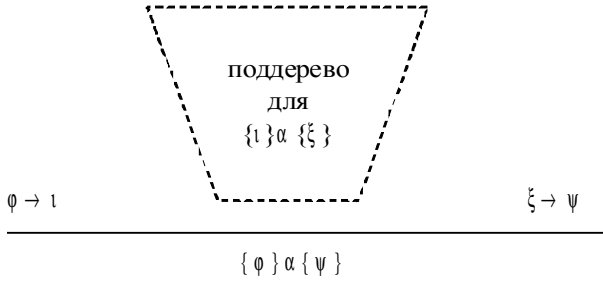
Выберем произвольное дерево вывода высоты не более $(h+1)$, удовлетворяющее условиям утверждения. Пусть корень этого дерева – формальная тройка $\{\phi\} \alpha \{\psi\}$. Если это дерево на самом деле имеет высоту не более h , то согласно предположению индукции, его корень $\{\phi\} \alpha \{\psi\}$ – истинная формальная тройка Хоара. Если же дерево имеет высоту $(h+1)$, то значит, его корень получился по одному из четырех правил вывода в аксиоматической семантике:

- по правилу усиления посылки/ослабления заключения PS/CW ,
- по правилу последовательной композиции SC ,
- по правилу недетерминированного выбора NC ,
- по правилу инварианта цикла LI .

Разберем все эти случаи по отдельности.

Случай применения правила PS/CW . Тогда выбранное дерево имеет следующий вид:

⁹² Здесь используется формальная семантика языка HeMo, в которой тест не имеет никакого побочного эффекта.

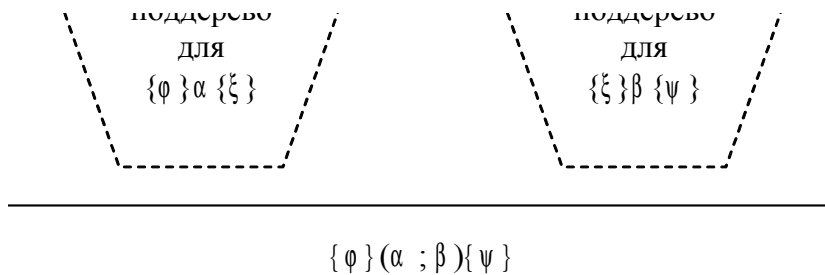


где типизированные формулы $(\varphi \rightarrow \iota)$ и $(\xi \rightarrow \psi)$ являются истинными, а поддерево вывода для $\{\iota\} \alpha \{\xi\}$ имеет высоту не более h , все его листья – или истинные типизированные формулы, или истинные формальные тройки Хоара для языка HeMo, или аксиомы системы AS. По предположению индукции тройка $\{\iota\} \alpha \{\xi\}$ является истинной. Поэтому для любой пары состояний $u, w \in SP$, если $u \langle \alpha \rangle w$, то имеет место следующее:

$$\begin{aligned}
 u \models \varphi &\Rightarrow (\text{т.к. } \varphi \rightarrow \iota \text{ истинная формула}) \Rightarrow \\
 &\Rightarrow u \models \iota \Rightarrow (\text{т.к. } \{\iota\} \alpha \{\xi\} \text{ истинная тройка}) \Rightarrow \\
 &\Rightarrow w \models \xi \Rightarrow (\text{т.к. } \xi \rightarrow \psi \text{ истинная формула}) \Rightarrow w \models \psi.
 \end{aligned}$$

В силу произвольного выбора состояний u и w заключаем, что формальная тройка $\{\varphi\} \alpha \{\psi\}$ тоже истинна.

Случай применения правила SC. Тогда выбранное дерево имеет следующий вид:

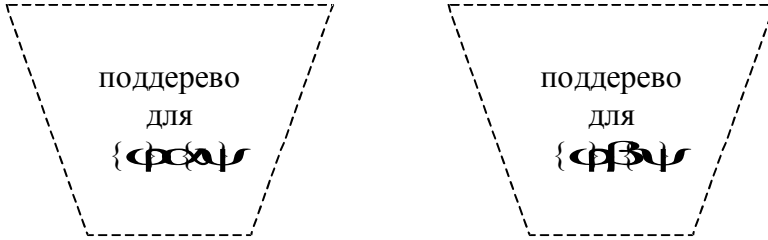


где поддеревья вывода для $\{\varphi\}\alpha\{\xi\}$ и $\{\xi\}\beta\{\psi\}$ имеет высоту не более h , все их листья – или истинные типизированные формулы, или истинные формальные тройки Хоара для языка HeMo, или аксиомы системы AS. По предположению индукции обе тройки является истинными. Поэтому для любой пары состояний $u, w \in SP$, если $u \langle \alpha ; \beta \rangle w$, то существует⁹³ такое состояние $v \in SP$, что $u \langle \alpha \rangle v$ и $v \langle \beta \rangle w$. Поэтому для таких u , w и v имеет место следующее:

$$u \models \varphi \Rightarrow (\text{т.к. } \{\varphi\}\alpha\{\xi\} \text{ истинная тройка}) \Rightarrow \\ \Rightarrow v \models \xi \Rightarrow (\text{т.к. } \{\xi\}\beta\{\psi\} \text{ истинная тройка}) \Rightarrow w \models \psi.$$

В силу произвольного выбора состояний u и w заключаем, что формальная тройка $\{\varphi\}(\alpha ; \beta)\{\psi\}$ тоже истинна.

Случай применения правила NC. Тогда выбранное дерево имеет следующий вид:



$$\{\varphi\}\alpha\beta\{\psi\}$$

где поддеревья вывода для $\{\varphi\}\alpha\{\psi\}$ и $\{\varphi\}\beta\{\psi\}$ имеет высоту не более h , все их листья – или истинные типизированные формулы, или истинные формальные тройки Хоара для языка HeMo, или аксиомы системы AS. По предположению индукции обе тройки является истинными. Поэтому для любой пары состояний $u, w \in SP$, если $u \langle \alpha \cup \beta \rangle w$, то⁹⁴ $u \langle \alpha \rangle w$ или $u \langle \beta \rangle w$. Поэтому имеет место следующее:

⁹³ Здесь используется формальная семантика $(\alpha;\beta)$ в языке HeMo; согласно формальной семантике невозможно войти внутрь блока или выйти изнутри блока.

⁹⁴ Здесь используется формальная семантика $(\alpha\cup\beta)$ в языке HeMo; согласно формальной семантике невозможно войти внутрь блока или выйти изнутри блока.

В силу произвольного выбора состояний u и w заключаем, что формальная тройка $\{\varphi\}(\alpha; \beta)s\{\psi\}$ тоже истинна.

поддерево
для
 $\{1, 2, 3\}$


$$\begin{aligned} u \models \varphi &\Rightarrow (\text{т.к. } \varphi \text{ совпадает с } i, \text{ а } u - c v_0) \Rightarrow \\ &\Rightarrow v_0 \models i \Rightarrow (\text{т.к. } \{i\}\alpha\{i\} \text{ истинная тройка}) \Rightarrow \\ &\Rightarrow v_1 \models i \Rightarrow (\text{т.к. } \{i\}\alpha\{i\} \text{ истинная тройка}) \Rightarrow \\ &..... \\ &\Rightarrow v_{(n-1)} \models i \Rightarrow (\text{т.к. } \{i\}\alpha\{i\} \text{ истинная тройка}) \Rightarrow \\ &\Rightarrow v_n \models i \Rightarrow (\text{т.к. } i \text{ совпадает с } \psi, \text{ а } v_n - c w) \Rightarrow w \models \psi. \end{aligned}$$

На этом доказательство утверждения закончено. ■

28

Для удобства введем следующее техническое

Определение. Дерево вывода в аксиоматической семантике AS языка НеМо выводит формальную тройку Хоара, если эта тройка – корень этого дерева. Дерево вывода в аксиоматической семантике AS формальных троек Хоара для языка НеМо называется правильным, если все его листья – или аксиомы AS, или истинные формальные тройки, или истинные типизированные формулы. Правильное дерево вывода в аксиоматической семантике AS формальных троек Хоара будем называть доказательством⁹⁶ в аксиоматической семантике AS, когда все его листья – или аксиомы AS, или истинные типизированные формулы. Доказательство в аксиоматической семантике AS языка НеМо доказывает формальную тройку Хоара, если эта тройка – это корень этого дерева.

С учетом новых технических терминов формулируется так: всякое правильное дерево вывода в аксиоматической семантике AS выводит истинную формальную тройку Хоара для языка НеМо. В частности, всякое доказательство в аксиоматической семантике AS языка НеМо доказывает истинную формальную тройку.

Из следует, что все позитивные примеры формальных троек Хоара для языка НеМо, которые были рассмотрены в лекции 17, оказались истинными не случайно: в лекции 16 они были снабжены доказательствами в аксиоматической семантике AS.

Из этого же утверждения следует, что для двух отрицательных примеров из лекции 18, в которых обсуждались не истинные тройки⁹⁷, нельзя построить ни доказательства, ни правильного дерева вывода в аксиоматической семантике AS.

Однако ничего не говорит о возможности (или невозможности) построить доказательство в аксиоматической семантике AS языка НеМо для отри-

⁹⁶ Конечно же, это определение «доказательства в аксиоматической семантике AS» несколько не согласуется с понятием «доказательства в аксиоматической системе AS», если последнее понимать в том строгом смысле, в каком понятие доказательства в аксиоматической системе было введено в лекции 4. У доказательства в аксиоматической системе AS все листья должны быть аксиомами. А у доказательства в аксиоматической семантике среди листьев могут быть истинные типизированные формулы (что выясняется каким-то «внешним» для программ способом). Однако эту несогласованность легко устранить, если ввести дополнительную «схему аксиом» предметной области DOM: θ , где θ - любая истинная типизированная формула. Тогда можно сказать, что доказательство в аксиоматической семантике AS – это доказательство в аксиоматической системе AS, расширенной аксиомами предметной области, заданными схемой DOM.

⁹⁷ Напомним, эти две тройки были получены из истинной тройки №3 (см. лекцию 17) в результате изменения предусловия и, соответственно, постусловия.

цательного примера истинной формальной тройки Хоара из предыдущей лекции 17, которая осталась истинной после внесения ошибки в программу. Точно также утверждение ничего не говорит о возможности построить доказательства в аксиоматической семантике AS для следующих истинных троек Хоара

- $\{\text{FALSE}\} \alpha \{\psi\}$,
- $\{\phi\} \alpha \{\text{TRUE}\}$,
- $\{\phi\} \text{LOOP} \{\psi\}$,

где ϕ и ψ - произвольные типизированные формулы, а LOOP – произвольная никогда не останавливающаяся НеМо-программа. Или более общий вопрос: для всякой ли истинной формальной тройки Хоара $\{\rho \ \phi\} \delta \ \alpha \ \{\psi\}$ можно построить доказательство⁹⁸ в аксиоматической семантике AS? Для того что бы ответить на этот вопрос в следующей лекции 19, в этой лекции необходимо ввести одно важное понятие и доказать дополнительное утверждение.

Слабейшее предусловие

Определение 15

Для любой вычислительной программы⁹⁹ α , и любого постусловия⁹⁹ ψ , слабейшее предусловие для α и ψ это такое условие¹⁰⁰ $wp(\alpha, \psi)$, что имеет место следующие два свойства.

- $wp(\alpha, \psi)$ является «корректным предусловием для α и ψ », то есть $\{wp(\alpha, \psi)\} \alpha \{\psi\}$ – истинное утверждение частичной корректности.
- $wp(\alpha, \psi)$ является «самым слабым» корректным предусловием для α и ψ , то есть для любого предусловия ϕ , если $\{\phi\} \alpha \{\psi\}$ – истинное утверждение частичной корректности, то ϕ всегда влечет $wp(\alpha, \psi)$ (то есть формула $(\phi \rightarrow wp(\alpha, \psi))$ является истинной).

Из определения следует, что с точностью до эквивалентности слабейшее предусловие единственно в следующем смысле. Если $wp'(\alpha, \psi)$ и $wp''(\alpha, \psi)$ – два «разных» предусловия, то оба они – самые слабые, то есть $wp'(\alpha, \psi) \Rightarrow wp''(\alpha, \psi)$ и $wp''(\alpha, \psi) \Rightarrow wp'(\alpha, \psi)$, и следовательно $wp'(\alpha, \psi) \Leftrightarrow wp''(\alpha, \psi)$.

⁹⁸ Правильное дерево вывода для истинной формальной тройки Хоара автоматически существует: оно состоит только из корня в котором стоит сама эта истинная тройка.

⁹⁹ Так как это общее определение, то речь идет об общем понятии, а не понятии, имеющим смысл только для языка НеМо. Поэтому программы и постусловия – не обязательно НеМо-программы и не обязательно типизированные формулы.

¹⁰⁰ « wp » - от «Weakest Precondition».

Иногда, однако, имеет смысл различать форму представления слабейшего предусловия и тогда можно говорить о разных предусловиях¹⁰¹.

На первый взгляд может показаться, что «построить» хотя бы одно слабейшее предусловие всегда возможно следующим образом.

Пусть α - вычислительная программа, а ψ - постусловие. Обозначим через $\text{PRE}(\alpha, \psi)$ «множество¹⁰²» всех корректных предусловий для α и ψ :

$$\begin{aligned} \text{PRE}(\alpha, \psi) &\equiv \\ &\equiv \{ \varphi - \text{условие: } \{ \varphi \} \alpha \{ \psi \} - \text{истинное утверждение частичной корректности} \}. \end{aligned}$$

Тогда в качестве $\text{wp}(\alpha, \psi)$ можно было бы принять $\text{wp}(\alpha, \psi) \equiv \bigvee_{\varphi \in \text{PRE}(\alpha, \psi)} \varphi$ дизъюнкцию всех условий из $\text{PRE}(\alpha, \psi)$.

Но даже если на время забыть о том, что еще надо доказать, что $\text{PRE}(\alpha, \psi)$ — это множество, и что надо конкретизировать язык представления условий φ в этом «множестве», то все равно предложенный вариант построения слабейшего предусловия не состоятелен, так как он подразумевает возможность «построить» дизъюнкцию возможно бесконечного множества условий!

Например, пусть α - тривиальная программа $\text{VAR } x : \text{INT}; x := 0$ на языке HeMo, а ψ - условие, представленное простой формулой $x=0$. Но даже в этом случае среди корректных предусловий будут все следующие формулы

$$\dots, x=-3, x=-2, x=-1, x=0, x=1, x=2, \dots, x=100392, \dots$$

Можно заметить, что вообще любая типизированная формула будет корректным предусловием для таких α и ψ .

¹⁰¹ Ситуация похожа, например, на ситуацию с тавтологиями логики первого порядка. С точностью до эквивалентности все эти формулы можно представить единственной формуле TRUE. Но тем не менее изучать и использовать имеет смысл разные тавтологии.

¹⁰² Слово «множество» взято в кавычки, потому что нет уверенности, что перед нами именно множество, а не совокупность вроде «множества всех множеств», которую обсуждалось в лекции 3. Кроме того, его следующее «формальное» определение ни чуть не лучше «формального» определения «множества всех компьютеров», которое обсуждалось в той же лекции 3: так как не определен язык записи «условий», то это понятие так же неопределено, как и понятие «компьютер».

Поэтому (если ограничиваться рамками языка HeMo) $PRE(\alpha, \psi) = \{ \varphi : \varphi$ - типизированная формула, в которой переменная x типа INT не имеет связанных вхождений $\}$. И если не задумываться о синтаксической корректности языка предусловий, то можно представить слабейшее предусловие в следующем виде

$$wp(\alpha, \psi) = \bigvee_{\substack{\varphi \text{ - типизированная формула,} \\ \text{в которой } x \text{ типа INT} \\ \text{не имеет связанных вхождений}}} \varphi.$$

Но ведь «построенное» выражение, во-первых, не является формулой языка HeMo (так как содержит бесконечную дизъюнкцию), и, вообще говоря, вовсе не построено явно (так как явно выписать бесконечную дизъюнкцию невозможно).

Сказанное выше отнюдь не означает невозможность существования слабейшего предусловия для всех программ и постусловий. Оно даже не означает невозможность построения слабейшего предусловия для определенных классов программ и постусловий. Так, в частности, слабейшим предусловием для HeMo-программы $\alpha \equiv (\text{VAR } x : \text{INT}; x := 0)$ и HeMo-постусловия $\psi \equiv (x=0)$ возможное слабейшее предусловие $wp(\alpha, \psi)$ – это просто TRUE. Действительно, формальная тройка Хоара для языка HeMo $\{\text{TRUE}\}(\text{VAR } x : \text{INT}; x := 0) \{x=0\}$ является истинной, и импликация $(\varphi \Rightarrow \text{TRUE})$ также является истиной для любой формулы φ .

Утверждение 17

Пусть α - произвольная программа виртуальной HeMo-машины. Пусть V – множество ее переменных α , а δ - совокупности описаний этих переменных в α . Для каждой переменной¹⁰³ X из V , пусть T_X – ее тип в соответствии с δ , S_X – пусть новая (уникальная для каждой переменной X) переменная с описанием $\text{VAR } S_X : (\text{INT ARRAY OF } T_X)$. Пусть, кроме того, L – новая уникальная переменная с описанием $\text{VAR } L : (\text{INT ARRAY OF INT})$. Пусть τ - совокупность описаний всех новых переменных. Для любого состояния $t \in SP(\delta\tau)$, для любого целого числа k пусть

- $t(V)_k$ – это состояние из $SP(\rho\delta)$, которое на всякой переменной $X \in V$ совпадает¹⁰⁴ с $t(S_X)(k)$,

¹⁰³ Используются заглавные буквы для обозначения «метаварiableных» для переменных, так как если бы были использованы строчные (например « x »), то в соответствии с синтаксисом языка виртуальной HeMo-машины, это были бы конкретные переменные (переменная « x » в частности).

¹⁰⁴ Напомним, что семантика переменной S_X типа $(\text{INT ARRAY OF } T_X)$ есть частичная функция из $t(S_X) : \mathbf{Z} \rightarrow D_{T_X}$ заданная наследственно-конечной таблицей. Поэтому $t(S_X)(k) = \text{APL}(t(X), k)$

- а $t(L)_k$ – это пусть $(t(L))(k)$.

Тогда для любых состояний $s \in SP(\rho\delta)$ и $t \in SP(\rho\delta\tau)$, для любого целого числа k и любой метки M , которая встречается в α , имеет место следующие свойства.

1. $(t(L)_k, s)$ является конфигурацией, в которой контроль в α находится в метке $M \Leftrightarrow t \models (L[k] = M)$. В частности:
 - 1.1. $(t(L)_k, s)$ – начальная конфигурация $\alpha \Leftrightarrow t \models (L[k] = 0)$,
 - 1.2. $(t(L)_k, s)$ – заключительная конфигурация $\alpha \Leftrightarrow t \models \bigvee_{n \in \text{END}} (L[k] = n)$, где END – множество всех меток, которые встречаются в α , но не меняют ни одного оператора.
2. Состояния $t(V)_k$ и s совпадают $\Leftrightarrow t \models \bigwedge_{X \in V} (S_X[k] = X)$. Кроме того, если $t(V)_k$ и s совпадают, то для любой формулы χ , в которой все свободные переменные описаны в δ , но не используются переменные описанные в τ , имеет место эквивалентность: $s \models \chi \Leftrightarrow t \models \chi'$, где χ' получилась в результате замены каждой переменной $X \in V$ на $S_X[k]$.
3. Для любого оператора присваивания « $M: X := E \text{ goto NEXT}$ » в α : $((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{s_{(k+1)}}))$ является срабатыванием этого оператора \Leftrightarrow имеет место конъюнкция двух условий
 - 3.1. $t \models (L[k] = M)$, $t \models \bigvee_{n \in \text{NEXT}} (L[k+1] = n)$,
 - 3.2. $t \models \bigvee_{Y \in V \setminus \{X\}} (S_Y[k+1] = S_Y[k])$ и $t \models (S_X[k+1] = E')$, где E' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[k]$.
4. Для любого условного оператора « $M: \text{if } \chi \text{ then NEXT}^+ \text{ else NEXT}$ » в α : $((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{(k+1)}))$ является срабатыванием этого оператора \Leftrightarrow имеет место конъюнкция двух условий
 - 4.1. $t \models (L[k] = M)$, $t \models \bigvee_{Y \in V} (S_Y[k+1] = S_Y[k])$,
 - 4.2. если $t \models \chi'$, то $t \models \bigvee_{n \in \text{NEXT}^+} (L[k+1] = n)$, иначе $t \models \bigvee_{n \in \text{NEXT}} (L[k+1] = n)$, где χ' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[k]$.

Доказательство проведем отдельно для пункта.

1. $(t(L)_k, s)$ является конфигурацией, в которой контроль в α находится в метке $M \Leftrightarrow t(L)_k = M \Leftrightarrow t \models (L[k] = M)$. В частности:
 - 1.1. $(t(L)_k, s)$ – начальная конфигурация $\alpha \Leftrightarrow t(L)_k = M \Leftrightarrow t \models (L[k] = 0)$,
 - 1.2. $(t(L)_k, s)$ – заключительная конфигурация $\alpha \Leftrightarrow t(L)_k \in \text{END} \Leftrightarrow t \models \bigvee_{n \in \text{END}} (L[k] = n)$.

2. Состояния $t(V)_k$ и s совпадают $\Leftrightarrow s \in SP(\delta)$ совпадает с состоянием $t(V)_k \in SP(\rho\delta)$, которое на всякой переменной $X \in V$ равно $(t(S_X))(k) \Leftrightarrow t \models_{\wedge_{X \in V}} (S_X[k] = X)$. Теперь приступим к доказательству¹⁰⁵ второй части пункта 2. Так как в формуле χ все свободные переменные описаны в δ , но не используются переменные описанные в τ , то в χ нет кванторов по новым переменным (которые как раз и описаны в τ), и ни одно вхождение ни одной переменной $X \in V$ не является связанным каким-либо квантором. Поэтому при подстановке вместо каждой переменной $X \in V$ соответствующего выражения $S_X[k]$ в χ происходит замена свободных вхождений переменной $X \in V$ на выражение $S_X[k]$ ¹⁰⁶. Пусть ξ получается из χ в результате замены всякой переменной $X \in V$ на ее значение $s(X)$, а ξ' получаются из χ' в результате замены всякого выражения $S_X[k]$ (где $X \in V$) на его значение $(t(S_X))(k)$. В силу сказанного ранее о «свободе» всех вхождений всех переменных $X \in V$ в χ имеем: $s \models \chi \Leftrightarrow s \models \xi$. Аналогично предыдущему рассуждению, $t \models \xi' \Leftrightarrow t \models \chi'$ (силу сказанного ранее о свободе в χ' всех вхождений всех переменных S_X , где $X \in V$). Кроме того, так как в ξ нет ни одного вхождения ни одной свободной переменной, то $s \models \xi \Leftrightarrow t \models \xi$. Следовательно, если $t(X)_k$ и s совпадают, то (по уже доказанному) $t \models_{\wedge_{X \in V}} (S_X[k] = X)$ и, следовательно, ξ и ξ' синтаксически совпадают. Поэтому имеем следующую цепочку эквивалентностей: $s \models \chi \Leftrightarrow s \models \xi \Leftrightarrow t \models \xi \Leftrightarrow t \models \xi' \Leftrightarrow t \models \chi'$ и, как следствие, эквивалентность $s \models \chi \Leftrightarrow t \models \chi'$.
3. Для любого оператора присваивания «M: $X := E$ goto NEXT» в α имеем (см. определение срабатывания помеченного оператора присваивания в лекции 12): $((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{(k+1)}))$ является срабатыванием этого оператора $\Leftrightarrow ((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{(k+1)}))$ – это пара конфигураций программы α такая, что $t(L)_k = M$, $t(L)_{(k+1)} \in \text{NEXT}$, а $t(V)_{(k+1)}$ отличается от $t(V)_k$ только значением переменной X , которое в $t(V)_{(k+1)}$ равно $t(V)_k(E) \Leftrightarrow t \models^{107} (L[k] = M)$, $t \models^{108} \bigvee_{n \in \text{NEXT}} (L[k+1] = n)$, $t \models^{108} \bigvee_{Y \in$

¹⁰⁵ Вообще говоря, для равенства справедлив так называемый принцип Лейбница, который на неформально-философском языке формулируется следующим образом: «Смысл любого утверждения не изменится, если в нем равное заменено на равное»; поэтому на неформально-философском уровне эквивалентность $s \models \chi \Leftrightarrow t \models \chi'$ является частным случаем принципа Лейбница (при условии, что $t(V)_k$ и s совпадают). Однако на формально-математическом уровне строгости следует быть более аккуратным в рассуждениях.

¹⁰⁶ Напомним, что S_X – переменная, а k – целое число (то есть величина представленная своей десятичной записью), а не переменная и не параметр.

¹⁰⁷ В силу уже доказанного пункта утверждения.

¹⁰⁸ В силу уже доказанного пункта утверждения.

$\forall_{\{X_i\}} (S_Y[k+1] = S_Y[k])$ и $t \models 108 (S_X[k+1] = E')$, где E' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[k]$.

4. Для любого условного оператора «M: if χ then $NEXT^+$ else $NEXT^-$ » в α имеем (см. определение срабатывания помеченного оператора присваивания в лекции 12): $((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{(k+1)}))$ является срабатыванием этого оператора $\Leftrightarrow ((t(L)_k, t(V)_k), (t(L)_{(k+1)}, t(V)_{(k+1)}))$ – это пара конфигураций программы α , для которой выполнено или $t(L)_k = M$, $t(L)_{(k+1)} \in NEXT^+$, $t(V)_{(k+1)} = t(V)_k$ и $t(V)_k \models \chi$, или $t(L)_k = M$, $t(L)_{(k+1)} \in NEXT^-$, $t(V)_{(k+1)} = t(V)_k$ и неверно $t(V)_k \models \chi \Leftrightarrow t \models 107 (L[k] = M)$, $t \models 108 \vee_{Y \in V} (S_Y[k+1] = S_Y[k])$, и если $t \models 108 \chi'$, то $t \models 107 \vee_{n \in NEXT^+} (L[k+1] = n)$, иначе – $t \models 107 \vee_{n \in NEXT^-} (L[k+1] = n)$, где χ' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[k]$. ■

Из предыдущего Утверждение 17 и утверждения 2 из лекции 14 (о корректности алгоритма трансляции вычислительных HeMo-программ в вычислительные программы виртуальной HeMo-машины) получается следующее утверждение о существовании слабейшего предусловия.

Утверждение 18

Пусть β - произвольная вычислительная HeMo-программа. Пусть $V \equiv X_1, \dots, X_m$ – список (без повторов) всех переменных и параметров β , а δ – совокупности описаний этих переменных. Пусть ρ - произвольная совокупность описаний параметров, совместная по переменным с β . Пусть ψ - произвольная формула, совместимая по типам переменных с ρ и β . Для каждой переменной или параметра X_i ($i \in [1..m]$) пусть T_i – ее тип в соответствии с ρ , S_i – пусть новая¹⁰⁹ (уникальная для каждой переменной X) переменная с описанием $VAR S_i : (INT \text{ ARRAY OF } T_i)$. Пусть, кроме того, L – новая¹⁰⁹ уникальная переменная с описанием $VAR L : (INT \text{ ARRAY OF } INT)$. Пусть K и N – новые уникальные переменные типа INT . Пусть τ - совокупность описаний всех новых¹¹⁰ переменных. Тогда существует формула WP , все свободные переменные которой описаны в $\rho\delta$, все связанные – в τ или в формуле ψ , такая, что

1. $\{\rho \text{ } WP\} \beta \{\psi\}$ истинная формальная тройка Хоара для языка HeMo;
2. для любой формальной тройки Хоара для языка HeMo $\{\rho \text{ } \phi\} \beta \{\psi\}$, в которой не используются новые переменные (описанные в τ), если $\models \{\rho \text{ } \phi\} \alpha \{\psi\}$, то $(\phi \rightarrow WP)$ – формула истинная в любом состоянии из $SP(\rho \text{ } \delta)$.

¹⁰⁹ То есть не описанная ни в α , ни в ρ , ни в ψ .

¹¹⁰ Включая K и N .

Доказательство. Пусть выполнены условия утверждения. Транслируем программу β в программу α виртуальной НеМо-машины с эквивалентной вычислительной семантикой в соответствии с алгоритмом, описанным в лекции 14. В соответствии с утверждение 2 из этой же лекции, программа α использует только те же переменные и параметры, что и β .

Тогда для любого состояния $s \in SP(\rho\delta)$ имеем:

- для любого состояния $s' \in SP(\rho\delta)$, если $s \langle \beta \rangle s'$ то $s' \models \psi \Leftrightarrow$
- \Leftrightarrow для любого состояния $s' \in SP(\rho\delta)$, если $s \langle \beta \rangle s'$ то $s' \models \psi \Leftrightarrow$
- \Leftrightarrow для любой конечной полной трассы α , которая начинается с конфигурации с состоянием s , в последнем состоянии этой трассы $s' \models \psi \Leftrightarrow$
- \Leftrightarrow для любого натурального $n \geq 0$ и любой последовательности $(l_0, s_0), \dots (l_n, s_n)$ конфигураций β , если $l_0 = 0, s_0 = s, l_n$ не метит ни одного оператора в α , и для любого $k \in [0..(n-1)]$ пара конфигураций $(l_k, s_k)(l_{k+1}, s_{k+1})$ – срабатывание какого либо помеченного оператора в α , то $s_n \models \psi \Leftrightarrow^{111}$
- $\Leftrightarrow s \models \forall \text{VAR } N : \text{INT}; \forall \text{VAR } L : (\text{INT ARRAY OF INT});$
 $\forall \text{VAR } S_1 : (\text{INT ARRAY OF } T_1); \dots \forall \text{VAR } S_m : (\text{INT ARRAY OF } T_m);$
 $((L[0] = 0 \wedge (S_1[0] = X_1 \wedge \dots \wedge S_m[0] = X_m) \wedge \bigvee_{n \in \text{END}} (L[N] = n) \wedge$
 $\bigvee \text{VAR } K : \text{INT}; (0 \leq K < N \rightarrow (\bigvee_{op \in \text{ASG}(\alpha)} F(\text{op}, K_s) \vee \bigvee_{op \in \text{CND}(\alpha)} F(\text{op}, K_s)))$
 $\rightarrow \psi')$ где
- ψ' получается из ψ в результате замены каждой переменной X_1, \dots, X_m на выражение $S_1[N], \dots, S_m[N]$ соответственно;
- дизъюнкция $\bigvee_{op \in \text{ASG}(\alpha)} F(\text{op}, K_s)$ взята по всем операторам присваивания в α , и для каждого оператора присваивания «M: X:= E goto NEXT» в α формула $F(\text{M: X:= E goto NEXT}, K)$ – это конъюнкция $(L[K] = M) \wedge (\bigvee_{n \in \text{NEXT}} (L[K+1] = n)) \wedge (\bigvee_{Y \in V \setminus \{X\}} (S_Y[K+1] = S_Y[K]) \wedge (S_X[K+1] = E')$, а E' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[K]$;
- дизъюнкция $\bigvee_{op \in \text{CND}(\alpha)} F(\text{op}, K_s)$ взята по всем условным операторам в α , и для каждого оператора присваивания «M: if χ then NEXT⁺ else NEXT⁻» в α формула $F(\text{M: if } \chi \text{ then NEXT}^+ \text{ else NEXT}^-, K)$ – это конъюнкция $(L[K] = M) \wedge (\bigvee_{Y \in V} (S_Y[k+1] = S_Y[k])) \wedge (\chi' \rightarrow \bigvee_{n \in \text{NEXT}^+} (L[K+1] = n)) \wedge (\neg \chi' \rightarrow \bigvee_{n \in \text{NEXT}^-} (L[K+1] = n))$, а χ' получилась в результате замены каждой переменной $Z \in V$ на $S_Z[K]$.

¹¹¹ В соответствии с Утверждение 17.

Обозначим всю последнюю формулу (целиком) WP. Согласно эквивалентности, доказанной одновременно с построением этой формулы WP, для любого состояния $s \in SP(\rho\delta)$ следующее равносильно:

$$s \models WP \Leftrightarrow \text{для любого состояния } s' \in SP(\rho\delta), \text{ если } s \langle \beta \rangle s' \text{ то } s' \models \psi.$$

Поэтому формула WP имеет следующие свойства.

1. Для любого состояния $s \in SP(\rho\delta)$, если $s \models WP$, то по доказанной эквивалентности $s' \models \psi$ в любом состоянии $s' \in SP(\rho\delta)$ таком что $s \langle \beta \rangle s'$, то есть $\models \{ \rho \text{ WP} \} \beta \{ \psi \}$.
2. Пусть для какой-либо формулы ϕ , в которой не используются новые переменные описанные в τ , имеет место $\{ \rho \phi \} \beta \{ \psi \}$. Тогда для любого состояния $s \in SP(\rho\delta)$, $s \models \phi$ влечет, что для любого состояния $s' \in SP(\rho\delta)$ из $s \langle \beta \rangle s'$ следует $s' \models \psi$. Но по доказанной эквивалентности, если для любого состояния $s' \in SP(\rho\delta)$ из $s \langle \beta \rangle s'$ следует $s' \models \psi$, то $s \models WP$. Поэтому $s \models (\phi \rightarrow WP)$. в любом состоянии из $SP(\rho\delta)$, то есть $(\phi \rightarrow WP)$ – истинная формула. ■

Можно заметить, что доказательство Утверждение 18 о слабейшем предусловии носит (как говорят в таких случаях) «конструктивный характер», то есть доказано не только существование некоего объекта (примера слабейшего предусловия в данном случае), но и предложен метод, как этот объект построить. Этот метод, описанный в утверждении, действительно можно описать как алгоритм¹¹² построения по произвольной HeMo-программе α (с параметрами заданными описаниями ρ) и произвольному постусловию ψ для этой программы слабейшего предусловия $WP(\rho\alpha, \psi)$.

Тогда вместо любой формальной тройки Хоара для языка HeMo $\{ \rho \phi \} \alpha \{ \psi \}$ задача проверки/опровержения ее истинности сводилась бы к задаче проверки/опровержения истинности «беспрограммной» формулы $(\phi \rightarrow WP(\rho\alpha, \psi))$, а следующее дерево вывода в аксиоматической семантике AS

$$\frac{\{ WP(\rho\alpha, \psi) \} \alpha \{ \psi \}}{\{ \rho \phi \} \alpha \{ \psi \}}$$

¹¹² Задание для самостоятельной работы: описать этот алгоритм и оценить его сложность (в зависимости от размеров программы, описаний параметров и постусловия).

было бы правильным тогда и только тогда, когда $\models \{ \rho \phi \} \alpha \{ \psi \}$ (в силу Утверждение 18).

Однако, такой путь совершенно не практичный, так как приводит к повышению порядка логики, формулы которой надо проверять. Действительно, для каждой переменной X , которая используется в программе, метод предлагает завести новую переменную S_X ; если тип переменной X был T_X , то тип новой переменной S_X уже $(\text{INT ARRAY OF } T_X)$; поэтому, если множество значений X было множество B_X , то множество значений S_X уже подмножество $(D_X)^Z$, состоящее из всех функции с конечной областью определенности из целых чисел в D_X . Так, например, если переменная X была целого типа, то новая переменная S_X будет иметь тип целочисленного массива с целыми индексами. Поэтому надеяться на «легкость» проверки/опровержения «беспрограммной» формулы $(\phi \rightarrow \text{WP}(\rho\alpha, \psi))$ к сожалению не приходится.

Но все-таки есть два важных случая, когда слабейшее предусловие имеет смысл представить явно.

Утверждение 19

Пусть ρ и δ - произвольные (совместные) описания параметров и программных переменных, ϕ и ψ - произвольные формулы, в которых все свободные переменные описаны в $\rho\delta$.

1. Если оператор присваивания « $X := E$ » совместим с описаниями $\rho\delta$, то имеем: формальная тройка Хоара для языка НеМо $\{ \rho \phi \} \delta(X := E) \{ \psi \}$ является истинной тогда и только тогда, когда формула $(\phi \rightarrow \psi_{E/X})$ истинна во всех состояниях $\text{SP}(\rho\delta)$.
2. Если тест $(\chi?)$ совместим с описаниями $\rho\delta$, то имеем: формальная тройка Хоара для языка НеМо $\{ \rho \phi \} \delta(\chi?) \{ \psi \}$ является истинной тогда и только тогда, когда формула $(\phi \rightarrow (\chi \rightarrow \psi))$ истинна во всех состояниях $\text{SP}(\rho\delta)$.

Доказательство.

Для большей ясности мы опускаем все описания переменных, но предполагаем, что все формулы, программы и формальные тройки синтаксически корректны.

1. $\models \{ \phi \} (X := E) \{ \psi \} \Leftrightarrow$ для любого состояния $s \in \text{SP}(\rho\delta)$, если $s \models \phi$ то $\text{upd}(s, X, s(E)) \models \psi$. Пусть $d = s(E)$ – это значение выражения E в состоянии s . Пусть ψ' получается из ψ в результате замены переменной X на значе-

ние d . Так как все вхождения X в ψ свободны, то имеем $\text{upd}(s, X, s(E)) \models \psi \Leftrightarrow \text{upd}(s, X, s(E)) \models \psi' \Leftrightarrow s \models \psi'$ (т.к. ψ' не имеет вхождений переменной X). Но в формуле $\psi_{E/X}$ свободны все вхождения всех переменных, которые встречаются в E . Поэтому $s \models \psi' \Leftrightarrow s \models \psi_{E/X}$, так как ψ' можно представить как результат замены всех вхождений (которые стоят на месте X) выражения E в $\psi_{E/X}$ на $d=s(E)$. является истинной тогда и только тогда, когда формула $(\phi \rightarrow \psi_{E/X})$ истинна во всех состояниях $SP(p\delta)$. Следовательно, $\models \{\phi\}(X:=E)\{\psi\} \Leftrightarrow$

\Leftrightarrow для любого $s \in SP(p\delta)$, если $s \models \phi$ то $\text{upd}(s, X, s(E)) \models \psi \Leftrightarrow$
 \Leftrightarrow для любого $s \in SP(p\delta)$, если $s \models \phi$ то $\text{upd}(s, X, s(E)) \models \psi' \Leftrightarrow$
 \Leftrightarrow для любого $s \in SP(p\delta)$, если $s \models \phi$ то $s \models \psi' \Leftrightarrow$
 \Leftrightarrow для любого $s \in SP(p\delta)$, если $s \models \phi$ то $s \models \psi_{E/X} \Leftrightarrow$
 \Leftrightarrow для любого $s \in SP(p\delta)$, $s \models (\phi \rightarrow \psi_{E/X}) \Leftrightarrow$
 $\Leftrightarrow \models (\phi \rightarrow \psi_{X/E})$.

2. $\models \{\phi\}(\chi?)\{\psi\} \Leftrightarrow$ для любого состояния $s \in SP(p\delta)$, если $s \models \phi$ то если $s \models \chi$ то $s \models \psi \Leftrightarrow$ для любого состояния $s \in SP(p\delta)$, $s \models (\phi \rightarrow (\chi \rightarrow \psi)) \Leftrightarrow \models (\phi \rightarrow (\chi \rightarrow \psi))$. ■

В силу уже доказанного о надежности аксиоматической семантики, из только что доказанного Утверждение 19 следует, что

1. $\psi_{E/X}$ является слабейшим предусловием для присваивания $(X:=E)$ и постусловия ψ (так как $\{\psi_{E/X}\}(X:=E)\{\psi\}$ аксиома надежной системы AS);
2. $(\chi \rightarrow \psi)$ является слабейшим предусловием для теста $(\chi?)$ и постусловия ψ (т.к. $\{\chi \rightarrow \psi\}(\chi?)\{\psi\}$ аксиома надежной системы AS).

Лекция 19. Полнота аксиоматической семантики. Идеализированный язык Pascal и его аксиоматическая семантика

Полнота аксиоматической семантики языка HeMo

Следующее утверждение формализует полноту аксиоматической семантики SA для языка HeMo: любая истинная формальная тройка Хоара для HeMo имеет доказательство в аксиоматической семантике AS.

Утверждение 20

Для любой истинной формальной тройки Хоара для языка HeMo существует дерево вывода в аксиоматической семантике AS для языка HeMo, все листья которого истинные типизированные формулы или аксиомы системы AS.

База индукции соответствует случаю, когда программа – это или присваивание, или тест.

- $$\frac{\{ \vdash_{\text{EX}} \} \quad \{ \vdash_{\text{EX}} (Y = E) \}}{\{ \vdash_{\text{EX}} (Y = E) \}} \begin{array}{l} \text{(аксиома A)} \\ \text{(правило PS/CW)} \end{array}$$

- $$\frac{\frac{\{ \langle \varphi \rangle \}}{\{ \langle \varphi \rangle \}} \quad \{ \langle \varphi \rangle \}}{\{ \langle \varphi \rangle \}} \quad \text{(аксиома Т)}$$
- $$\frac{\{ \langle \varphi \rangle \}}{\{ \langle \varphi \rangle \}} \quad \text{(правило PS/CW)}$$

29

Теперь сделаем индукционное предположение, что для любой программы π состоящей из не более чем $k \geq 0$ конструкций «;», « \cup » и «*», для любых формул χ и ξ , если (синтаксически корректная) формальная тройка Хорара $\{\chi\}\pi\{\xi\}$ истинна, то существует ее доказательство в аксиоматической семантике AS.

Шаг индукции состоит в том, что для любой программы α состоящей из $(k+1)$ конструкцию «;», « \cup » и «*», для любых формул ϕ и ψ , надо доказать, что истинность тройки $\{\phi\}\alpha\{\psi\}$ влечет ее доказуемость в аксиоматической семантике AS. Для этого надо рассмотреть все варианты устройства программы α : (1) $\alpha \equiv (\beta ; \gamma)$, (2) $\alpha \equiv (\beta \cup \gamma)$ и (3) $\alpha \equiv (\beta^*)$.

1. Пусть $\alpha \equiv (\beta ; \gamma)$ и $\vdash \{\phi\}\alpha\{\psi\}$. Согласно утверждению 3 из предыдущей лекции 18, для программы γ и постуловия ψ существует слабейшее предусловие ξ , которое совместимо по типам переменных с α , ψ и ϕ . Тогда начать доказательство истинной тройки $\{\phi\}\alpha\{\psi\}$ можно было бы следующим образом:

$$\frac{\{\phi\}\beta\{\xi\} \quad \{\xi\}\gamma\{\psi\}}{\{\phi\}(\beta ; \gamma)\{\psi\}} \quad (\text{правило SC})$$

Согласно утверждению 3 из лекции 18 тройка $\{\xi\}\gamma\{\psi\}$ является истинной (так как ξ - слабейшее предусловие для γ и ψ). Следовательно, по предположению индукции, существует доказательство этой тройки $\{\xi\}\gamma\{\psi\}$ в аксиоматической семантике AS.

Покажем, что тройка $\{\phi\}\beta\{\xi\}$ тоже является истинной. Для этого предположим противное, т.е. что эта тройка не является истинной, и, следовательно, существуют такие состояния $s', s'' \in SP(p\delta)$, что $s' \models \phi$, $s' \langle \beta \rangle s''$, но неверно $s'' \models \xi$. Так как $\vdash \{\xi\}\gamma\{\psi\}$ и не верно $s'' \models \xi$, то существует такое состояние $s''' \in SP(p\delta)$, что $s'' \langle \gamma \rangle s'''$ и неверно $s''' \models \psi$. Так как $s' \langle \beta \rangle s''$ и $s'' \langle \gamma \rangle s'''$, то $s' \langle (\beta ; \gamma) \rangle s'''$. Суммируя получаем: $s' \models \phi$, $s' \langle (\beta ; \gamma) \rangle s'''$ и неверно $s''' \models \psi$. – Противоречие с истинностью тройки $\{\phi\}(\beta ; \gamma)\{\psi\}$. Следовательно, тройка $\{\phi\}\beta\{\xi\}$ тоже является истинной и для нее по предположению индукции существует доказательство в аксиоматической семантике AS.

Следовательно, доказательство истинной формальной тройки Хорара $\{\phi\}\alpha\{\psi\}$ при $\alpha \equiv (\beta ; \gamma)$ можно начать с применения правила SC при-

няв в качестве сечения¹¹³ ξ слабое предусловие¹¹⁴ для программы γ и постусловия ψ , а потом построить доказательства для истинных троек с более простыми программами $\{\phi\}\beta\{\xi\}$ и $\{\xi\}\gamma\{\psi\}$.

2. Пусть $\alpha \equiv (\beta \cup \gamma)$ и $\vdash \{\phi\}\alpha\{\psi\}$. Тогда начать доказательство истинной тройки $\{\phi\}\alpha\{\psi\}$ можно было бы следующим образом:

$$\frac{\{\phi\}\beta\{\psi\} \quad \{\phi\}\gamma\{\psi\}}{\{\phi\}(\beta \cup \gamma)\{\psi\}} \text{ (правило NC)}$$

Обе тройки $\{\phi\}\beta\{\psi\}$ и $\{\phi\}\gamma\{\psi\}$ в посылках этого правила являются истинными. Действительно, для любых состояний $s', s'' \in SP(\rho\delta)$ если $s' \langle \beta \rangle s''$ или $s' \langle \gamma \rangle s''$, то $s' \langle (\beta \cup \gamma) \rangle s''$. Так как $\vdash \{\phi\}\alpha\{\psi\}$, следовательно, если $s' \models \phi$, то $s'' \models \psi$. Поэтому если $s' \models \phi$ и $s' \langle \beta \rangle s''$ или $s' \langle \gamma \rangle s''$, то $s'' \models \psi$. В силу произвольного выбора состояний $s', s'' \in SP(\rho\delta)$ заключаем, что обе тройки $\{\phi\}\beta\{\psi\}$ и $\{\phi\}\gamma\{\psi\}$ истинны. По предположению индукции для них обоих существуют доказательства в аксиоматической семантике AS. Следовательно, доказательство истинной формальной тройки Хоара $\{\phi\}\alpha\{\psi\}$ при $\alpha \equiv (\beta \cup \gamma)$ можно начать с применения правила NC, а потом построить доказательства для истинных троек с более простыми программами $\{\phi\}\beta\{\psi\}$ и $\{\phi\}\gamma\{\psi\}$.

3. Пусть $\alpha \equiv (\beta^*)$ и $\vdash \{\phi\}\alpha\{\psi\}$. Согласно утверждению 3 из предыдущей лекции 18, для программы α и постусловия ψ существует слабое предусловие ι , которое совместимо по типам переменных с α , ψ и ϕ . Тогда начать доказательство истинной тройки $\{\phi\}\alpha\{\psi\}$ можно было бы следующим образом:

¹¹³ промежуточной формулы

¹¹⁴ которое существует согласно утверждению 3 из лекции 18

$$\begin{array}{c}
\frac{\{ \mathbf{i} \} \beta}{(\varphi \rightarrow \mathbf{i}) \rightarrow \{ \mathbf{i} \} \beta \quad (\mathbf{i} \rightarrow \varphi)} \quad \text{(правило LI)} \\
\frac{}{\{ \mathbf{i} \} \beta \quad \{ \mathbf{i} \} \beta \rightarrow \psi \quad (\mathbf{i} \rightarrow \varphi)} \quad \text{(правило PS/CW)}
\end{array}$$

Согласно утверждению 3 из лекции 18 формула $(\varphi \rightarrow \mathbf{i})$ является истинной (так как \mathbf{i} - слабое предусловие для (β^*) и ψ). Формула $(\mathbf{i} \rightarrow \psi)$ тоже является истинной т.к. для любого состояния $s \in SP(\rho\delta)$ всегда $s \models (\beta^*)s$, и следовательно, если $s \models \mathbf{i}$, то обязательно $s \models \psi$ (т.к. \mathbf{i} - слабое предусловие для (β^*) и ψ).

Теперь покажем, что тройка $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$ тоже является истинной. Для этого предположим противное, т.е. что эта тройка не является истинной, и, следовательно, существуют такие состояния $s', s'' \in SP(\rho\delta)$, что $s' \models \mathbf{i}$, $s' \models \beta s''$, но неверно $s'' \models \mathbf{i}$. Так как $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$ и не верно $s'' \models \mathbf{i}$, то¹¹⁵ существует такое состояние $s''' \in SP(\rho\delta)$, что $s'' \models (\beta^*)s'''$ и неверно $s''' \models \psi$. Так как $s' \models \beta s''$ и $s'' \models (\beta^*)s'''$, то $s' \models (\beta^*)s'''$. Суммируя, получаем: $s' \models \mathbf{i}$, $s' \models (\beta^*)s'''$ и неверно $s''' \models \psi$. – Противоречие с истинностью тройки $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$. Следовательно, тройка $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$ тоже является истинной и для нее по предположению индукции существует доказательство в аксиоматической семантике AS.

Следовательно, доказательство истинной формальной тройки Хоара $\{ \varphi \} \alpha \{ \psi \}$ при $\alpha \equiv (\beta^*)$ можно начать с применения правила PS/CW приняв в качестве сечения¹¹⁶ \mathbf{i} слабое предусловие¹¹⁷ для программы (β^*) и постусловия ψ , а потом применить к тройке $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$ правило LI, и наконец построить доказательства для истинной тройки с более простой программой $\{ \mathbf{i} \} \beta \{ \mathbf{i} \}$. ■

Таким образом, аксиоматическая семантика AS формальных троек Хоара для языка НеМо надежна (утверждение 1, лекция 18) и полна (настоящей лекции). Следовательно, чисто теоретически, вместо семантической проверки истинности формальных троек Хоара для языка НеМо мы (или компьютер) можем строить деревья вывода в аксиоматической системе AS

¹¹⁵ так как \mathbf{i} - слабое предусловие для (β^*) и ψ

¹¹⁶ инварианта

¹¹⁷ которое существует согласно утверждению 3 из лекции 19

в которых все листья – или аксиомы AS для присваиваний и тестов, или истинные «безпрограммные» формулы.

Идеализированный язык Pascal и его семантика

Определение 16

Идеализированный Pascal (I-Pascal) – это язык программирования с «абстрактными» типами данных¹¹⁸, в котором программы состоят из описаний переменных и тела, которое строится из операторов присваивания¹¹⁹ и условий¹²⁰ при помощи конструкций «;» для последовательной композиции, «IF-THEN-ELSE» для условного выбора и «WHILE-DO» для детерминированного цикла. Стандартная операционная семантика SOS для каждой i-Pascal-программы π , состоящей из описаний переменных δ и тела α , определяется на пространстве состояний $SP(\delta)$ телом α . Состояние¹²¹ – это произвольное отображение, которое сопоставляет каждой описанной переменной ее значение в соответствии с ее типом. Семантика¹²¹ программы π – это бинарное отношение на множестве состояний $[\pi] = [\alpha] \subseteq SP(\delta) \times SP(\delta)$. Для любых состояний $s', s'' \in SP(\delta)$ принято писать $s' \langle \pi \rangle s''$ или $s' \langle \alpha \rangle s''$ тогда и только тогда, когда $(s', s'') \in [\alpha]$ ($= [\pi]$). Таким образом, $[\pi] = [\alpha] = \{ (s', s'') \in SP(\delta) \times SP(\delta) : s' \langle \alpha \rangle s'' \}$. При фиксированной совокупности описаний δ , и при предположении, что все операторы присваивания и условия корректны относительно типов переменных в δ , бинарное отношение $[\alpha]$ определяется индукцией по структуре α следующим образом¹²².

1. Для оператора присваивания: $s' \langle X := E \rangle s'' \Leftrightarrow s'' = \text{UPD}(s', X, s'(E))$.
2. Для последовательной композиции: $s' \langle (\beta ; \gamma) \rangle s'' \Leftrightarrow s' \langle \beta \rangle s'''$ и $s''' \langle \gamma \rangle s''$ для некоторого состояния $s''' \in SP(\delta)$.
3. Для условного выбора: $s' \langle \text{IF } \varphi \text{ THEN } \beta \text{ ELSE } \gamma \rangle s'' \Leftrightarrow$ если в s' истинно (верно, выполнено, имеет место) условие φ ($s' \models \varphi$), то $s' \langle \beta \rangle s''$, иначе – $s' \langle \gamma \rangle s''$.

¹¹⁸ Слово «абстрактный» взято в кавычки, так как имеется в виду не тип данных, определенный в соответствии с нормами теории абстрактных типов данных (см. лекцию 11), а «неконкретизированный» тип данных, но который имеет свое множество значений и совокупность доступных операций. Примером такого «абстрактного» типа данных может служить тип СОЕД из задачи Э. Дейкстры о соединении черных и белых точек, которая обсуждалась в лекции 1.

¹¹⁹ При этом предполагается, что все присваивания синтаксически корректны и совместимы по типам данных.

¹²⁰ В качестве условий разрешается использовать отношения, то есть операции, которые определены для используемых типов данных, а возвращают значения булевского типа «И» и «Л». При этом предполагается, что все условия синтаксически корректны и совместимы по типам данных.

¹²¹ как и в языке HeMo

¹²² Используются макропеременные для переменных, выражений, условий и тел программ.

4. Для детерминированного цикла: $s' \langle \text{WHILE } \phi \text{ DO } \beta \rangle s'' \Leftrightarrow$ существует такое $k \geq 0$ и такая последовательность состояний $s_0, \dots, s_n \in \text{SP}(\delta)$, что $s_0 = s'$, $s_n = s''$, в состоянии s_n неверно условие ϕ (т.е. неверно $s \models \phi$), а для любого $i \in [0..(n-1)]$ имеет место $s_i \langle \beta \rangle s_{i+1}$.

Хороший пример программы на варианте языка I-Pascal, в котором допустим тип данных СОЕД, является программа, реализующая алгоритм Э. Дейкстры построения соединения «черных» и «белых» точек на плоскости, которая обсуждалась в лекции 1.

Следующее определение аксиоматической семантики языка I-Pascal близко к оригинальному определению, которое принадлежит А. Хоару и Н. Вирту¹²³.


Определение 17

Примем в качестве языка для предусловий и постусловий вариант языка логики, в котором все переменные типизированы типами языка I-Pascal. Тогда условие частичной корректности или тройка Хоара для I-Pascal – это тройка вида $\{ \rho \} \phi \} (\delta \alpha) \{ \psi \}$, где

- ρ - описания свободных переменных в формулах ϕ и ψ и параметров в теле программы α ,
- ϕ и ψ - формулы, которые не имеют коллизий связанных переменных, и называются предусловием и постусловием,
- $\delta \alpha$ - I-Pascal-программа с совокупность описаний программных переменных δ и телом α , в котором параметры, описанные в ρ , в операторах присваивания могут использоваться в правых частях.

Тройка $\{ \rho \} \phi \} (\delta \alpha) \{ \psi \}$ называется истинной (обозначение $\models \{ \rho \} \phi \} (\delta \alpha) \{ \psi \}$), когда для любых состояний $s', s'' \in \text{SP}(\rho\delta)$ если в состоянии $s' \models \phi$ и $s' \langle \alpha \rangle s''$ влечет $s'' \models \psi$. Тогда аксиоматическая семантика языка I-Pascal определяется следующим набором I-PAS схем правил вывода для троек:

Аксиома А присваивания (Assignment)

 обозначает результат замены всех вхождений переменной x в терм t

¹²³ Hoare C.A.R. and Wirth N. An Axiomatic Definition of the Programming Language PASCAL. Acta Informatica. 1973, v.2, pp.335-355.

Правило PS/CW усиления посылки и ослабления заключения
(Premise Strengthening/Conclusion Weakening)

$$\frac{\rho \delta(\phi \rightarrow \iota) \quad \{\rho \iota\} \delta \alpha \{\xi\} \quad \rho \delta(\xi \rightarrow \psi)}{\{\rho \phi\} \delta \alpha \{\psi\}}$$

Правило SC последовательной композиции (Sequential Composition)

$$\frac{\{\rho \phi\} \delta \alpha \{\xi\} \quad \{\rho \xi\} \delta \beta \{\psi\}}{\{\rho \phi\} \delta (\alpha ; \beta) \{\psi\}}$$

Правило CC условного выбора (Conditional Choice)

$$\frac{\{\rho (\phi \wedge \chi)\} \delta \alpha \{\psi\} \quad \{\rho (\phi \wedge (\neg \chi))\} \delta \beta \{\psi\}}{\{\rho \phi\} \delta (\text{IF } \chi \text{ THEN } \alpha \text{ ELSE } \beta) \{\psi\}}$$

Правило DL детерминированного цикла (Deterministic Loop)

$$\frac{\{\rho (\iota \wedge \chi)\} \delta \alpha \{\iota\}}{\{\rho \iota\} \delta (\text{WHILE } \chi \text{ DO } \alpha) \{\iota \wedge \neg \chi\}}$$

Так же, как в случае с аксиоматической семантикой AS языка HeMo, можно сэкономить на том, что совокупности описаний свободных и программных переменных (ρ и δ) не изменяются системой I-PAS. Следовательно, саму систему I-PAS удобнее сразу представить без описаний свободных и программных переменных, которая, собственно, исторически называется аксиоматической семантикой языка Pascal.

Правило PS/CW $\frac{\varphi \rightarrow 1 \quad \{1\} \alpha \{ \xi \} \quad \xi \rightarrow \psi}{\{ \varphi \} \alpha \{ \psi \}}$	Правило DL $\frac{\{1 \wedge \chi\} \alpha \{1\}}{\{1\} (\text{WHILE } \chi \text{ DO } \alpha) \{1 \vee \neg \chi\}}$
Правило SC $\frac{\{ \varphi \} \alpha \{ \xi \} \quad \{ \xi \} \beta \{ \psi \}}{\{ \varphi \} (\alpha ; \beta) \{ \psi \}}$	Правило CC $\frac{\{ \varphi \wedge \chi \} \alpha \{ \psi \} \quad \{ \varphi \wedge (\neg \chi) \} \beta \{ \psi \}}{\{ \varphi \} (\text{IF } \chi \text{ THEN } \alpha \text{ ELSE } \beta) \{ \psi \}}$
Аксиома* А $\frac{}{\{ \xi_{t/x} \} (x := t) \{ \xi \}}$	

(* : $\xi_{t/x}$ обозначает результат замены всех вхождений переменной x в ξ на терм t)

Аналогично тому, как было доказано утверждение 1 из лекции 18 о надежности аксиоматической семантики AS языка HeMo, можно доказать следующую теорему надежности для языка I-Pascal.

Теорема 1

Если в каком-либо дереве вывода в аксиоматической семантике I-PAS для идеализированного языка Pascal все листья – истинные формулы, или истинные формальные тройки Хоара, или аксиомы системы I-PAS, то и корень этого дерева вывода – истинная тройка Хоара для языка I-Pascal.

Доказательство – см. доказательство утверждения 1 из лекции 18. ■

Удивительным оказывается то, что для языка I-Pascal и системы I-PAS не имеет места аналог из данной лекции 18 о полноте аксиоматической семантики AS языка HeMo. И дело здесь не в том, что пропущено какое-либо важное правило вывода, а в том, что без дополнительной гипотезы о допустимых типах данных в языке I-Pascal доказать теорему полноты невозможно. Эта гипотеза может быть сформулирована следующим образом.

Определение 18 (гёделезуемости¹²⁴ данных)

Пусть T – тип данных некоторого языка программирования. Будем говорить, что T гёделизуем, когда средствами логики с типами данных этого языка программирования есть возможность представить при помощи фиксированного набора переменных любые конечные последовательности значений типа T , и на этом представлении реализуемы «операции»

1. `valid` – проверки, представляет ли аргумент какую-либо последовательность,
2. `length` – вычисления длины закодированной последовательности,
3. `element` – доступа к любому элементу последовательности.

Будем говорить, что язык программирования допускает гёделизацию данных (или имеет гёделизуемые типы данных), он имеет тип, изоморфный целым числам с операцией следования¹²⁵ и неравенствами, и если для любой тип данных этого языка гёделизуем.

НеМо имеет гёделизуемые типы данных. Действительно, пусть T – произвольный тип данных НеМо, а D – его множество значений. Тогда для любого целого числа k , любую конечную последовательность¹²⁶ $d_0, \dots, d_k \in D$ значений этого типа можно представлять парой, состоящей из переменной целого типа `INT` для хранения числа k и массива типа `(INT ARRAY OF T)` «в котором значения d_0, \dots, d_k хранятся в ячейках с 0-ой по k -ую»¹²⁷. Программно на языке НеМо такое представление поддерживается следующим образом.

Пусть K и S новые переменные¹²⁸ типов `INT` и `(INT ARRAY OF T)` соответственно. Тогда функции, существование которых требуется в определении гёделизуемости, можно задать, например, так.

¹²⁴ Термин «гёделезуемость» введен только в этом курсе, так как в научной литературе на русском языке нет соответствующего понятия. Наш выбор термина обусловлен связью с гёделевскими номерами, кодирующими последовательности натуральных чисел.

¹²⁵ то есть операцией « $+1$ »

¹²⁶ эта последовательность пуста при $k < 0$

¹²⁷ Фраза про хранение значений в ячейках массива взята в кавычки, так как семантика массивов – это функции, и, следовательно, правильнее вместо заковыченной фразы было бы сказать, что «значение которого d_0 в точке 0, \dots d_k – в точке k ».

¹²⁸ Используются метоперменные K и S для переменных языка НеМо, вместо которых могут использоваться любые подходящие настоящие переменные языка НеМо. Термин «новые» означает, что конкретные переменные для K и S не встречаются в программе, в формуле, в которых надо гёделизовать последовательности.

1. Если $\text{valid} \equiv \forall \text{VAR } I : \text{INT}; (0 \leq I \leq K \rightarrow S[I] \neq \omega?)$, то для любого состояния $s \in \text{SP}$ имеем: $s \models \text{valid} \Leftrightarrow s(K) < 0$ или $s(K) \geq 0$ и все элементы¹²⁹ $s(S)(0), \dots, s(S)(s(K))$ определены.
2. Length – это просто текущее значения переменной K .
3. Element – это просто текущее значение элемента массива S .

Слабейшее предусловие в языке I-Pascal

Доказательство теоремы полноты для языка I-Pascal могло бы начаться с аналога утверждения 3 из предыдущей лекции 18 о существовании слабейшего предусловия, но построенного уже без явной гёделизации с использованием массивов, а с использованием гёделизации данных в терминах операций `code`, `valid`, `length` и `element`. Однако такой подход потребовал бы необходимость разработать алгоритм трансляции I-Pascal-программ, доказать его корректность. Поэтому мы используем другой подход для конструирования слабейшего предусловия, разработанный¹³⁰ Э. Дейкстрой (который, кстати, может быть перенесен на язык HeMo). В нашем описании этого подхода мы предполагаем, что среди типов данных есть тип, изоморфный типу `INT` языка HeMo, что выбраны и зафиксированы совокупность описаний свободных переменных (параметров) ρ и совокупность описаний программных переменных δ , а все тела программ и формулы – совместимы с этими описаниями (и не имеют коллизий связанных переменных).

Суть подхода – описание рекурсивного алгоритма, который по телу программы и формулы-постусловию строит формулу - слабейшее предусловие для программы с данным телом и данного постусловия.

Алгоритм $\text{WP}(\text{первый_аргумент}, \text{второй_аргумент})$

- [
1. `первый_аргумент` – синтаксически правильная I-Pascal-программа, в которой все переменные описаны в δ , а все параметры – в ρ ;
 2. `второй_аргумент` – синтаксически правильная формула, в которой все свободные переменные описаны в $\rho\delta$
-]// Предусловие алгоритма.
1. Если `первый_аргумент` – это оператор присваивания « $X := E$ », а `второй_аргумент` – формула ψ , то $\text{WP} := \psi_{E/X}$.
 2. Если `первый_аргумент` – это программа $(\beta ; \gamma)$, а `второй_аргумент` – формула ψ , то $\text{WP} := \text{WP}(\beta, \text{WP}(\gamma, \psi))$.

¹²⁹ То есть функция $s(S)$ определена во всех точках интервала $[0..s(K)]$.

¹³⁰ Дейкстра Э. Дисциплина программирования. – М.: Мир, 1979

3. Если первый_аргумент – это программа IF χ THEN β ELSE γ , а второй_аргумент – формула ψ , то $WP := (\chi \wedge WP(\beta, \psi)) \vee ((\neg\chi) \wedge WP(\gamma, \psi))$.
4. Если первый_аргумент – это программа WHILE χ DO β , а второй_аргумент – формула ψ , то пусть
 - 4.1. S – «набор» новых переменных, представляющих конечные последовательности значений программных переменных¹³¹, а K и I – новые переменные типа INT;
 - 4.2. $valid(S)$ – формула, выражающая, что представленные посредством переменных в S значения являются последовательностями значений программных переменных в $\rho\delta$;
 - 4.3. $length(S, K)$ – формула, выражающая, что все представленные посредством переменных в S имеют длину равную $K \geq 0$;
 - 4.4. $VAR = element(S, I)$ – формула, выражающая, что каждая программная переменная принимает значение, равное I -ому элементу соответствующей ей последовательности;
 - 4.5. $COPY$ набор новых переменных, каждая из которых соответствует какой-либо программной переменной, а $VAR = COPY$ – формула, выражающая равенство каждой программной переменной и ее напарника из $COPY$;
 - 4.6. для любой формулы ϕ , все свободные переменные которой описаны в $\rho\delta$, $\phi_{element(S, I)/VAR}$ обозначает результат подстановки вместо каждой программной переменной, I -ого элемента соответствующей ему последовательности;
 - 4.7. для любой формулы ϕ , все свободные переменные которой описаны в $\rho\delta$, $\phi_{element(S, I)/COPY}$ обозначает результат подстановки вместо напарника из $COPY$ каждой программной переменной, I -ого элемента соответствующей ему последовательности.

Тогда¹³² $WP :=$

$$\begin{aligned}
 & \forall S \forall K ((valid(S) \wedge length(S, K) \wedge VAR = element(S, 0) \wedge (\neg\chi) \\
 & \quad elemsmt(S, K)/VAR \wedge \\
 & \quad \wedge \forall I (0 \leq I < K \rightarrow \\
 & \quad \rightarrow \chi_{elemsmt(S, I)/VAR} \wedge ((WP(\beta, VAR = COPY))_{element(S, I+1)/COPY})_{elemsmt(S, I)/VAR})) \\
 & \rightarrow \Psi_{elemmt(S, K)/VAR}).
 \end{aligned}$$

¹³¹ то есть переменных описанных в δ

¹³² Здесь мы отступаем от строго следованию синтаксису формул логики с типизированными переменными, так как задаем лишь «схему» формулы.

[WP такая формула, что все ее свободные переменные описаны в $\rho\delta$ и для любой тройки Хоара для языка I-Pascal $\{\rho \ \phi\}(\delta \ \beta)\{\psi\}$, тройка $\{\rho \ \phi\} \alpha\{\psi\}$ истинна тогда и только тогда, когда формула $(\phi \rightarrow WP)$ истинна в любом состоянии из $SP(\rho\delta)$.
] // Постусловие.

Утверждение 21

Алгоритм WP является totally корректным относительно своих предусловия и постусловия.

Доказательство.

Алгоритм WP всегда завершает свою работу, если входные данные удовлетворяют предусловию алгоритма. Действительно, пусть α и ψ - произвольные такие тело программы и формула, для которых надо исполнить алгоритм WP. Это исполнение можно представить в виде дерева рекурсивных вызовов, корень которого – $WP(\alpha, \psi)$, узлы которого – это все вызовы $WP(\beta, \xi)$, которые (рекурсивно) возникали при исполнении $WP(\alpha, \psi)$, а дуги – это все пары $((WP(\beta', \xi'), WP(\beta'', \xi''))$ такие, что при исполнении вызова $WP(\beta', \xi')$ непосредственно (т.е. не рекурсивно) возник вызов $WP(\beta'', \xi'')$. Заметим, что на каждом вызове $WP(\beta, \xi)$ могут возникнуть не более двух рекурсивных вызовов (в случае, когда тело построено при помощи последовательной композиции или условного выбора), и каждый рекурсивный вызов происходит с первым аргументом, содержащей меньше конструкций «;», «IF-THEN-ELSE» и «WHILE-DO» чем β . Поэтому это дерево имеет конечную степень ветвления (не более 2), и каждая его ветвь имеет конечную длину (не более числа конструкций «;», «IF-THEN-ELSE» и «WHILE-DO» чем α). По лемме Кёнига это дерево рекурсивных вызовов конечно. Следовательно, вычисление $WP(\alpha, \psi)$ завершается. В силу произвольного выбора α и ψ , удовлетворяющих предусловию, завершаемость алгоритма доказана.

Осталось показать частичную корректность алгоритма. Это можно сделать индукцией по числу конструкций «;», «IF-THEN-ELSE» и «WHILE-DO» в первом аргументе, то есть в теле программы.

База индукции соответствует случаю, когда первый аргумент – это какое-либо присваивание «X:= E». В этом случае постусловие алгоритма можно считать доказанными, т.к. оно полностью соответствует пункту 1 ут-

верждения 4 из предыдущей лекции 18, где доказано, что $\Psi_{E/X}$ – слабое предусловие для присваивания « $X := E$ ».

Предположение индукции состоит в том, что алгоритм частично корректен для всякого тела программы, в котором не более $k \geq 0$ конструкций «;», «IF-THEN-ELSE» и «WHILE-DO».

Шаг индукции состоит в разборе случаев, соответствующих конструкциям «;», «IF-THEN-ELSE» и «WHILE-DO» в первом аргументе вызова $WP(\alpha, \psi)$.

Случай, когда α имеет вид $(\beta ; \gamma)$. По предположению индукции для γ , все свободные переменные $WP(\gamma, \psi)$ описаны в $\rho\delta$. Следовательно, согласно предположению индукции для β , все свободные переменные $WP(\beta, WP(\gamma, \psi))$ тоже описаны в $\rho\delta$. Но $WP(\alpha, \psi) \equiv WP(\beta, WP(\gamma, \psi))$. Далее, пусть ϕ – произвольное предусловие (согласованное с описаниями $\rho\delta$). Тогда имеем: $\vdash \{\phi\} \alpha \{\psi\} \Leftrightarrow$ для любых состояний $s', s'' \in SP(\rho\delta)$, если $s \models \phi$ и $s' \langle \alpha \rangle s''$, то $s'' \models \psi \Leftrightarrow$ для любых состояний $s', s'' \in SP(\rho\delta)$, если $s \models \phi$ и существует некое состояние s''' , для которого $s' \langle \beta \rangle s'''$ и $s''' \langle \gamma \rangle s''$, то $s'' \models \psi \Leftrightarrow^{133}$ для любых состояний $s', s''' \in SP(\rho\delta)$, если $s \models \phi$ и $s' \langle \beta \rangle s'''$, то $s''' \models WP(\gamma, \psi) \Leftrightarrow^{134}$ для любого состояния $s' \in SP(\rho\delta)$, если $s \models \phi$, то $s' \models WP(\beta, WP(\gamma, \psi)) \Leftrightarrow (\phi \rightarrow WP(\alpha, \psi))$ – истинная формула.

Случай, когда α имеет вид IF χ THEN β ELSE γ . По предположению индукции для β и γ , все свободные переменные $WP(\beta, \psi)$ и $WP(\gamma, \psi)$ описаны в $\rho\delta$. Все свободные переменные χ тоже описаны в $\rho\delta$ (т.к. χ – часть тела программы). Но $WP(\alpha, \psi) \equiv (\chi \wedge WP(\beta, \psi)) \vee ((\neg\chi) \wedge WP(\gamma, \psi))$. Следовательно, все свободные переменные $WP(\alpha, \psi)$ тоже описаны в $\rho\delta$. Далее, пусть ϕ – произвольное предусловие (согласованное с описаниями $\rho\delta$). Тогда имеем: $\vdash \{\phi\} \alpha \{\psi\} \Leftrightarrow$ для любых состояний $s', s'' \in SP(\rho\delta)$, если $s \models \phi$ и $s' \langle \alpha \rangle s''$, то $s'' \models \psi \Leftrightarrow$ для любых состояний $s', s'' \in SP(\rho\delta)$, если $s \models \phi$ и или $s' \models \chi$ и $s' \langle \beta \rangle s''$, или $s' \models (\neg\chi)$ и $s' \langle \gamma \rangle s''$, то $s'' \models \psi \Leftrightarrow^{135}$ для любого состояния $s' \in SP(\rho\delta)$, если $s \models \phi$, то или $s' \models \chi$ и $s' \models WP(\beta, \psi)$, или $s' \models (\neg\chi)$ и $s' \models WP(\gamma, \psi) \Leftrightarrow (\phi \rightarrow WP(\alpha, \psi))$ – истинная формула.

¹³³ согласно предположению индукции для γ и в силу эквивалентности $(\exists _ \Rightarrow _) \Leftrightarrow \forall (_ \Rightarrow _)$

¹³⁴ согласно предположению индукции для β

¹³⁵ согласно предположению индукции для β и γ

Случай, когда α имеет вид WHILE χ DO β .

1. В подформулах $\text{valid}(S)$ и $\text{length}(S, K)$ свободными являются переменные S и K . Но в формуле $\text{WP}(\alpha, \psi)$ эти подформулы находятся в области действия кванторов $\forall S$ и $\forall K$. Поэтому в формуле $\text{WP}(\alpha, \psi)$ в подформулах $\text{valid}(S)$ и $\text{length}(S, K)$ не остается свободных переменных.
 1. В подформуле $\text{VAR}=\text{element}(S, 0)$ свободными являются программные переменные, описанные в δ , и переменные S . Но в формуле $\text{WP}(\alpha, \psi)$ эта подформула находится в области действия кванторов $\forall S$. Поэтому в формуле $\text{WP}(\alpha, \psi)$ в подформуле $\text{VAR}=\text{element}(S, 0)$ все свободные переменные – это программные переменные, описанные в δ .
 2. По предположению индукции для β , свободные переменные формулы $(\text{WP}(\beta, \text{VAR}=\text{COPY}))$ – это все переменные, описанные в $\rho\delta$, и напарники программных переменных COPY . Однако после подстановки $(\text{WP}(\beta, \text{VAR}=\text{COPY}))_{\text{element}(S, I+1)/\text{COPY}}$ место переменных COPY занимают « $(I+1)$ -ые элементы переменных S ». Далее, после еще одной подстановки $((\text{WP}(\beta, \text{VAR}=\text{COPY}))_{\text{element}(S, I+1)/\text{COPY}})_{\text{elemsmt}(S, I)/\text{VAR}}$ место программных переменных занимают « I -ые элементы переменных S ». Но в формуле $\text{WP}(\alpha, \psi)$ все вхождения « $(I+1)$ -ых элементов переменных S » и « I -ых элементов переменных S » находятся в области действия кванторов $\forall S$ и $\forall I$, поэтому они являются связанными, а свободными после этой квантификации в подформуле $((\text{WP}(\beta, \text{VAR}=\text{COPY}))_{\text{element}(S, I+1)/\text{COPY}})_{\text{elemsmt}(S, I)/\text{VAR}}$ остаются только вхождения параметров, описанных в ρ .
 3. В формуле ψ свободными являются все переменные, описанные в $\rho\delta$. В подформуле $\psi_{\text{elemsmt}(S, K)/\text{VAR}}$ место программных переменных занимают « K -ые элементы переменных S ». Но в формуле $\text{WP}(\alpha, \psi)$ все вхождения « K -ых элементов переменных S » находятся в области действия кванторов $\forall S$ и $\forall K$, поэтому они являются связанными, а свободными после этой квантификации в подформуле $\psi_{\text{elemsmt}(S, K)/\text{VAR}}$ остаются только вхождения параметров, описанных в ρ .
- Суммируя, получаем: в формуле $\text{WP}(\alpha, \psi)$ все свободные переменные описаны в $\rho\delta$.

Далее, пусть ϕ - произвольное предусловие (согласованное с описаниями $\rho\delta$). Тогда имеем: $\vdash \{\phi\} \alpha \{\psi\} \Leftrightarrow$

\Leftrightarrow для любых состояний $s', s'' \in \text{SP}(\rho\delta)$, если $s' \vdash \phi$ и $s' \langle \alpha \rangle s''$, то $s'' \vdash \psi \Leftrightarrow$

- \Leftrightarrow для любых состояний $s', s'' \in SP(\rho\delta)$, если $s' \not\models \varphi$ и существуют $k \geq 0$ и последовательность состояний $s_0, \dots, s_k \in SP(\rho\delta)$ такая, что $s_0 = s', s_k = s'', s_k \models \chi$, а $s_i \not\models \chi$ и $s_i \langle \beta \rangle s_{i+1}$ для всех $i \in [0..(k-1)]$, то $s'' \models \psi \Leftrightarrow^{136}$
- \Leftrightarrow для любых состояний $s', s'' \in SP(\rho\delta)$, любого k и любой последовательности состояний $s_0, \dots, s_k \in SP(\rho\delta)$, если $s' \not\models \varphi$, $s_0 = s', s_k = s'', s_k \models \chi$, а $s_i \not\models \chi$ и $s_i \langle \beta \rangle s_{i+1}$ для всех $i \in [0..(k-1)]$, то $s'' \models \psi \Leftrightarrow$
- \Leftrightarrow для любого состояния¹³⁷ $s' \in SP(\rho\delta + S + K)$, если значения $s'(S)$ представляют последовательности какой-либо длины $s'(K) \geq 0$ значений программных переменных в некоторых состояниях из $SP(\rho\delta)$ таких, что
- $s' \not\models \varphi$, каждая программная переменная принимает в s' значение, равное 0-ому элементу соответствующей ей последовательности в $s'(S)$,
 - формула χ после подстановки вместо каждой программной переменной соответствующего ей значения из $(s'(S))(s'(K))$ становится ложной,
 - но для всех $i \in [0..(s'(K)-1)]$ формула χ после подстановки вместо каждой программной переменной соответствующего ей значения из $(s'(S))(i)$ остается истинной, а β начав вычисления при значениях программных переменных, равных значениям $(s'(S))(i)$, может остановиться только со значениями программных переменных равными $(s'(S))(i+1)$, тогда формула ψ после подстановки вместо каждой программной переменной соответствующего ей значения из $(s'(S))(s'(K))$ становится истинной
- \Leftrightarrow^{138}
- \Leftrightarrow для любого состояния $s' \in SP(\rho\delta)$, если $s' \not\models \varphi$,
то $s' \models \forall S \forall K ((\text{valid}(S) \wedge \text{length}(S, K) \wedge \text{VAR} = \text{element}(S, 0) \wedge$
 $\wedge (\neg \chi)_{\text{elemsmt}(S, K)/\text{VAR}} \wedge \forall I (0 \leq I < K \rightarrow$
 $\rightarrow \chi_{\text{elemsmt}(S, I)/\text{VAR}} \wedge$
 $\wedge ((\text{WP}(\beta, \text{VAR} = \text{COPY}))_{\text{element}(S, I+1)/\text{COPY}})_{\text{elemsmt}(S, I)/\text{VAR}})) \rightarrow$
 $\rightarrow \psi_{\text{elemsmt}(S, K)/\text{VAR}})$
- $\Leftrightarrow (\varphi \rightarrow \text{WP}(\alpha, \psi))$ – истинная формула.

¹³⁶ в силу эквивалентности $(\exists _ \Rightarrow _) \Leftrightarrow \forall (_ \Rightarrow _)$

¹³⁷ $(\rho\delta + S + K)$ обозначает совокупность описаний, которая получилась путем добавления к $\rho\delta$ описаний переменных из S (гёделизующих программные переменные), и целочисленной переменной K .

¹³⁸ так как формула φ не содержит вхождений переменных S и K , а в формуле $\text{WP}(\alpha, \psi)$ эти переменные связаны квантором всеобщности

Таким образом, частичная корректность алгоритма WP тоже доказана.

■

Полнота аксиоматической семантики языка I-Pascal

Аналог следующей теоремы полноты аксиоматической семантики идеализированного языка Pascal был впервые доказан¹³⁹ А.С. Куком в начале 1970-ых годов (и с тех пор теорема носит его имя).

Теорема 2

Если типы данных языка I-Pascal допускают гёделизацию, то для любой истинной тройки Хоара для языка I-Pascal существует дерево вывода в аксиоматической семантике I-PAS, все листья которого истинные типизированные формулы или аксиомы системы I-PAS.

Доказательство (набросок) в основном повторяет доказательство этой лекции, только слабейшее предусловие строится не в соответствии с утверждениями 3 и 4 предыдущей лекции 18, а в соответствии с Утверждение 21 данной лекции.

Доказательство проходит индукцией по структуре программы на языке I-Pascal. Пусть, как и в доказательстве, выбраны и зафиксированы произвольные описания параметров ρ и описания программных переменных δ .

База индукции соответствует случаю, когда программа — это или присваивание, и полностью повторяет доказательство для присваивания.

Теперь сделаем индукционное предположение, что для любой программы π состоящей из не более чем $k \geq 0$ конструкций «;», «IF-THEN-ELSE» и «WHILE-DO», для любых формул χ и ξ , если (синтаксически корректная) формальная тройка Хоара $\{\chi\}\pi\{\xi\}$ истинна, то существует ее доказательство в аксиоматической семантике AS.

Шаг индукции состоит в том, что для любой программы α состоящей из $(k+1)$ конструкцию «;», «IF-THEN-ELSE» и «WHILE-DO», для любых формул ϕ и ψ , надо доказать, что истинность тройки $\{\phi\}\alpha\{\psi\}$ влечет ее доказуемость в аксиоматической семантике I-PAS. Для этого надо рассмотреть все

¹³⁹ Cook S.A. Soundness and completeness of an axiom system for program verification. SIAM J. Comput. 1978, v.7, pp.70-90.

варианты устройства программы α : (1) $\alpha \equiv (\beta ; \gamma)$, (2) $\alpha \equiv \text{IF } \theta \text{ THEN } \beta \text{ ELSE } \gamma$ и (3) $\alpha \equiv \text{WHILE } \theta \text{ DO } \beta$. Случай (1), когда $\alpha \equiv (\beta ; \gamma)$ полностью повторяет доказательство для последовательной композиции. Поэтому остается рассмотреть два случая (2) и (3).

2. Пусть $\alpha \equiv (\text{IF } \theta \text{ THEN } \beta \text{ ELSE } \gamma)$ и $\vdash \{\phi\} \alpha \{\psi\}$. Тогда «начать» доказательство истинной тройки $\{\phi\} \alpha \{\psi\}$ можно было бы так:

$$\frac{\{\phi \wedge \theta\} \beta \{\psi\} \quad \{\phi \wedge (\neg \theta)\} \gamma \{\psi\}}{\{\phi\} (\text{IF } \theta \text{ THEN } \beta \text{ ELSE } \gamma) \{\psi\}} \text{ (правило CC)}$$

Обе тройки $\{\phi \wedge \theta\} \beta \{\psi\}$ и $\{\phi \wedge (\neg \theta)\} \gamma \{\psi\}$ в посылках этого правила являются истинными. Действительно, для любых состояний $s', s'' \in \text{SP}(\rho \delta)$ если $s' \models \theta$ и $s' \langle \beta \rangle s''$, или $s' \models (\neg \theta)$ и $s' \langle \gamma \rangle s''$, то $s' \langle \text{IF } \theta \text{ THEN } \beta \text{ ELSE } \gamma \rangle s''$. Так как $\vdash \{\phi\} \alpha \{\psi\}$, следовательно, если $s' \models \phi$, то $s'' \models \psi$. Поэтому если $s' \models \phi$, $s' \models \theta$ и $s' \langle \beta \rangle s''$, то $s'' \models \psi$. Аналогично, если $s' \models \phi$, $s' \models (\neg \theta)$ и $s' \langle \gamma \rangle s''$, то $s'' \models \psi$. В силу произвольного выбора состояний $s', s'' \in \text{SP}(\rho \delta)$ заключаем, что истинны обе тройки $\{\phi \wedge \theta\} \beta \{\psi\}$ и $\{\phi \wedge (\neg \theta)\} \gamma \{\psi\}$. По предположению индукции для них обоих существуют доказательства в аксиоматической семантике I-PAS. Следовательно, доказательство истинной формальной тройки Хоара $\{\phi\} \alpha \{\psi\}$ при $\alpha \equiv \text{IF } \theta \text{ THEN } \beta \text{ ELSE } \gamma$ можно начать с применения правила CC, а потом построить доказательства для истинных троек с более простыми программами $\{\phi \wedge \theta\} \beta \{\psi\}$ и $\{\phi \wedge (\neg \theta)\} \gamma \{\psi\}$.

3. Пусть $\alpha \equiv \text{WHILE } \theta \text{ DO } \beta$ и $\vdash \{\phi\} \alpha \{\psi\}$. Согласно Утверждение 21, для программы α и постусловия ψ существует слабое предусловие ι , которое совместимо по типам переменных с α , ψ и ϕ . Тогда «начать» доказательство истинной тройки $\{\phi\} \alpha \{\psi\}$ можно было бы следующим образом:

$$\frac{(\phi \rightarrow \iota) \quad \{\iota \wedge \theta\} \beta \{\iota\} \quad (\iota \wedge (\neg \theta) \rightarrow \psi)}{\{\phi\} (\text{WHILE } \theta \text{ DO } \beta) \{\psi\}} \begin{array}{l} \text{(комбинация} \\ \text{правил} \\ \text{DL и PS/WC)} \end{array}$$

Согласно Утверждение 21, формула $(\phi \rightarrow \iota)$ является истинной (так как ι - слабое предусловие для $\text{WHILE } \theta \text{ DO } \beta$ и ψ). Формула $(\iota \wedge (\neg \theta) \rightarrow \psi)$ тоже является истинной, так как для любого состояния $s \in \text{SP}(\rho \delta)$, если $s \models (\neg \theta)$, то $s \langle \text{WHILE } \theta \text{ DO } \beta \rangle s$, и следовательно, если $s \models (\iota \wedge (\neg$

θ)), то обязательно $s \models \psi$ (так как ι - слабое предусловие для $\text{WHILE } \theta \text{ DO } \beta$ и ψ). Теперь покажем, что тройка $\{\iota \wedge \theta\} \beta \{ \iota \}$ тоже является истинной. Для этого предположим противное, т.е. что эта тройка не является истинной, и, следовательно, существуют такие состояния $s', s'' \in \text{SP}(\rho\delta)$, что $s' \models \iota$, $s' \not\models \theta$ и $s' \langle \beta \rangle s''$, но неверно $s'' \models \iota$. Так как $\vdash \{ \iota \} (\text{WHILE } \theta \text{ DO } \beta) \{ \psi \}$ и не верно $s'' \models \iota$, то¹⁴⁰ существует такое состояние $s''' \in \text{SP}(\rho\delta)$, что $s'' \langle \text{WHILE } \theta \text{ DO } \beta \rangle s'''$ и неверно $s''' \models \psi$. Так как $s' \models \theta$, $s' \langle \beta \rangle s''$ и $s'' \langle \text{WHILE } \theta \text{ DO } \beta \rangle s'''$, то $s' \langle \text{WHILE } \theta \text{ DO } \beta \rangle s'''$. Суммируя получаем: $s' \models \iota$, $s' \langle \text{WHILE } \theta \text{ DO } \beta \rangle s'''$ и неверно $s''' \models \psi$. – Противоречие с истинностью тройки $\{ \iota \} (\text{WHILE } \theta \text{ DO } \beta) \{ \psi \}$. Следовательно тройка $\{\iota \wedge \theta\} \beta \{ \iota \}$ тоже является истинной и для нее по предположению индукции существует доказательство в аксиоматической семантике I-PAS.

Следовательно, доказательство истинной формальной тройки Хоара $\{\phi\} \alpha \{ \psi \}$ при $\alpha \equiv \text{WHILE } \theta \text{ DO } \beta$ можно начать с применения правила DL приняв в качестве инварианта ι слабое предусловие¹⁴¹ для программы $\text{WHILE } \theta \text{ DO } \beta$ и постусловия ψ , а затем построить доказательства для истинной тройки с более простой программой $\{\iota \wedge \theta\} \beta \{ \iota \}$.

■

Интересно заметить и то, что если язык НеМо ограничить только одним типом данных INT (то есть еще более усечь, чем диалект языка, принятый в лабораторных работах), то такой целочисленный НеМо тоже будет иметь гёделизуемые типы данных (точнее, единственный, но гёделизуемый тип данных). Именно этим свойством натуральных (и целых) чисел воспользовался К. Гёдель при доказательстве своей знаменитой теоремы неполноты, которая утверждает, что множество истинных формул арифметики натуральных или целых чисел не может быть аксиоматизировано¹⁴². Именно поэтому в этом курсе лекций выбран термин «гёделизуемость» для соответствующего понятия для любых типов данных.

¹⁴⁰ Так как ι - слабое предусловие для $\text{WHILE } \theta \text{ DO } \beta$ и ψ

¹⁴¹ которое существует согласно утверждению 3 из лекции 18

¹⁴² Заинтересованному читателю рекомендуется обратиться, например, к гл. 3 книги Мендельсон Э. Математическая логика. – М.: Наука, 1978.

Лекция 20. Основы верификации вычислительных программ (на примере языка HeMo и I-Pascal)

Понятие условий корректности

Аксиоматическая семантика языков HeMo и I-Pascal – это системы вывода для формальных троек Хоара этого языка, их синтаксис и семантика были определены в лекциях 16 и 17, а надежность и полнота были обоснованы в лекциях 18 и 19. Важную роль при этом играли понятие правильного дерева вывода и доказательства в аксиоматической семантике. Сейчас нам понадобится определить еще несколько важных понятий. Эти важные понятия будут определены для языков HeMo и I-Pascal, но могут быть соответствующим образом перенесены на другие языки программирования.

Определение. Квазидоказательство в аксиоматической семантике AS или I-PAS – это правильное дерево вывода в этой системе, у которого все листья – это или частные случаи аксиом, или формулы. Все формулы, которые встречаются в квазидоказательстве в качестве листьев, называются условиями корректности этого квазидоказательства.

Таким образом, квазидоказательство является доказательством в аксиоматической семантике, если все его условия корректности – истинные формулы. Пример квазидоказательства в аксиоматической системе I-PAS приведен на следующем рисунке.

$$\begin{array}{c}
\frac{(\varphi \rightarrow (l_{1/y})_{1/x}) \quad \frac{}{\{(l_{1/y})_{1/x}\}x:=1\{l_{1/y}\}}}{\frac{}{\{\varphi\}x:=1\{l_{1/y}\}}} \\
\uparrow \\
\frac{}{\{\varphi\} \gamma \{l\}} \quad \frac{}{\{l_{1/y}\} y:=1\{l\}} \quad \frac{(\{l \wedge (y \leq x)\} \beta' \{l'\}) \quad \frac{}{\{l'\}y:=y+2x+1\{l'\}}}{\frac{}{\{l \wedge (y \leq x)\} \beta \{l\}}} \quad \frac{}{\{l'\}x:=x+1\{l\}} \\
\frac{}{\{l\} \delta \{l \wedge \neg (y \leq x)\}} \quad \frac{(\{l \wedge \neg (y \leq x)\} \rightarrow \psi_{(x-1)/x}) \quad \frac{}{\{\psi_{(x-1)/x}\}x:=x-1\{\psi\}}}{\frac{}{\{l \wedge \neg (y \leq x)\} x:=x-1 \{\psi\}}} \\
\frac{\frac{}{\{\varphi\} \varepsilon \{l \wedge \neg (y \leq x)\}} \quad \frac{}{\{l \wedge \neg (y \leq x)\} x:=x-1 \{\psi\}}}{\frac{}{\{\varphi\} \alpha \{\psi\}}}
\end{array}$$

В этом примере α , β , γ , δ и ε - это следующее тело программы и его подтела на языке I-Pascal, в которых все переменные имеют тип INT:

Pascal (теорема 1, лекция 19) спецификация $\{z \geq 1\} \alpha \{x^2 \leq z \wedge (x+1)^2 > z\}$ является истинной. Фактически эта спецификация

утверждает, что программа α вычисляет в переменной x целую часть квадратного корня из положительного числа, заданного значением переменной z , а доказательство подтверждает это.

Для того, что бы автоматизировать процесс доказательства формализованных троек Хоара для программ на языках HeMo и I-Pascal от разработчика программы и ее спецификации (ее предусловия и постусловия) требуется аннотировать все циклы программы утверждениями, которые разработчик считает инвариантами, и которые остаются верными при каждой легальной¹⁴⁴ итерации тела соответствующего цикла. Для таких аннотированных троек Хоара для языка I-Pascal возникает понятие правильности, которое получается из понятия доказуемости (в аксиоматической семантике) и «аппроксимирует» понятие истинности тройки.

Определение. В аннотированной программе (на языке HeMo или I-Pascal) каждый цикл «*» и/или «while-do» аннотирован некоторой формулой, которая называется инвариантом этого цикла. В дальнейшем «*» или лексему «WHILE» цикла, аннотированного инвариантом i , будем записывать в виде «* i » и «WHILE i » соответственно. Всякую формализованную тройку Хоара с аннотированной программой мы будем называть аннотированной.

Определение. Доказательство аннотированной тройки (на языке HeMo или I-Pascal) называется доказательством правильности (в аксиоматической семантике AS или I-PAS соответственно), если в этом доказательстве все правила для цикла (LI и DL соответственно) применяются только к аннотированным циклам этой программы и используют инварианты этих циклов:

¹⁴⁴ Легальная итерация тела цикла для языка HeMo – это любая его итерация, а для языка I-Pascal – это итерация, когда условие цикла истинно.

Правило LI для аннотированного цикла *	Прав для аннотирован
$\frac{\{1\} \alpha \{1\}}{\{1\} (\alpha^{*1}) \{1\}}$	$\frac{\{1 \wedge \dots\}}{\{1\} (\text{WHILE}^1$

Определение. Аннотированная (формализованная) тройка (на языке НеМо или I-Pascal) называется правильной, если существует доказательство ее правильности в аксиоматической семантике (AS или I-PAS соответственно).

В частности, если цикл WHILE в разобранный выше примере аннотировать инвариантом $\imath \equiv (y=x^2 \wedge (x-1)^2 \leq z)$, то тройка $\{z \geq 1\} \alpha \{x^2 \leq z \wedge (x+1)^2 > z\}$ может служить примером правильной аннотированной тройки Хоара (что подтверждается доказательством на рисунке).

Утверждение 22

Всякая правильная аннотированная тройка является истинной.

Это утверждение является очевидным.

Определение. Условия корректности для аннотированной тройки Хоара $\{\phi\} \alpha \{\psi\}$ – это произвольное конечное множество VC формул такое, что тройка $\{\phi\} \alpha \{\psi\}$ является правильной тогда и только тогда, когда все утверждения из VC истинны.

Генерация условий корректности

Для автоматизации построения условий корректности для аннотированных троек Хоара для НеМо и I-Pascal определим два рекурсивных алгоритма AC¹⁴⁵ и VC¹⁴⁶. Оба алгоритма имеют два аргумента: первый аргумент – аннотированная программа на языке¹⁴⁷ НеМо или I-Pascal, а второй – формула. Алгоритм AC возвращает формулу, а алгоритм VC – конечное множество формул.

¹⁴⁵ «AC» от «Annotation Condition».

¹⁴⁶ «VC» от «Verification Condition».

¹⁴⁷ Мы будем иметь дело только с телом программы, предполагая, что совокупность описаний выбрана и зафиксирована.

Алгоритм AC:

- $AC(x:=t, \psi) = \psi_{t/x}$,
- $AC(\chi?, \psi) = (\chi \rightarrow \psi)$,
- $AC((\alpha ; \beta), \psi) = AC(\alpha, AC(\beta, \psi))$,
- $AC((\alpha \cup \beta), \psi) = AC(\alpha, \psi) \wedge AC(\beta, \psi)$,
- $AC(\text{IF } \chi \text{ THEN } \alpha \text{ ELSE } \beta, \psi) = ((\chi \wedge AC(\alpha, \psi)) \vee (\neg\chi \wedge AC(\beta, \psi)))$,
- $AC(\alpha^*, \psi) = \text{true}$,
- $AC(\text{WHILE } \chi \text{ DO } \alpha, \psi) = \text{true}$.

Алгоритм VC:

- $VC(x:=t, \psi) = \emptyset$,
- $VC(\chi?, \psi) = \emptyset$,
- $VC((\alpha ; \beta), \psi) = VC(\alpha, AC(\beta, \psi)) \cup VC(\beta, \psi)$,
- $VC((\alpha \cup \beta), \psi) = VC(\alpha, \psi) \cup VC(\beta, \psi)$,
- $VC(\text{IF } \chi \text{ THEN } \alpha \text{ ELSE } \beta, \psi) = VC(\alpha, \psi) \cup VC(\beta, \psi)$,
- $VC(\alpha^*, \psi) = VC(\alpha, \text{true}) \cup \{(\text{true} \rightarrow \psi)\}$,
- $VC(\text{WHILE } \chi \text{ DO } \alpha, \psi) = VC(\alpha, \text{true}) \cup \{((\text{true} \wedge \neg\chi) \rightarrow \psi), ((\text{true} \wedge \chi) \rightarrow AC(\alpha, \text{true}))\}$.

Приведем несколько примеров применения алгоритмов AC и VC. Для этого будем использовать аннотированную тройку

$$\{z \geq 1\} \\ (x:=1 ; y:=1 ; \text{WHILE } (y=x^2 \wedge (x-1)^2 \leq z) \text{ } y \leq z \text{ DO } (y:=y+2 \times x+1 ; x:=x+1) ; x:=x-1) \\ \{x^2 \leq z \wedge (x+1)^2 > z\}$$

на I-Pascal, рассмотренную в предыдущем пункте данной лекции. Если использовать уже введенные ранее обозначения, то имеем, например:

- $AC(x:=x-1, \psi) = \psi_{(x-1)/x} \equiv ((x-1)^2 \leq z \wedge ((x-1)+1)^2 > z)$,
- $AC(\delta, AC(x:=x-1, \psi)) = \text{true} \equiv (y=x^2 \wedge (x-1)^2 \leq z)$,
- $AC(\beta, \text{true}) = AC(y:=y+2 \times x+1, AC(x:=x+1, \text{true})) = (1_{(x+1)/x})_{(y+2 \times x+1)/y} \equiv ((y+2 \times x+1)=(x+1)^2 \wedge ((x+1)-1)^2 \leq z)$,
- $AC(\gamma, AC(\delta, AC(x:=x-1, \psi))) = AC(\gamma, \text{true}) = (1_{1/y})_{1/x} \equiv (1=1^2 \wedge (1-1)^2 \leq z)$,
- $AC(\alpha, \psi) = AC(\gamma, AC(\delta, AC(x:=x-1, \psi))) \equiv (1=1^2 \wedge (1-1)^2 \leq z)$,
- $VC(\alpha, \psi) = \{((\text{true} \wedge \neg(y \leq z)) \rightarrow \psi_{(x-1)/x}), ((\text{true} \wedge (y \leq z)) \rightarrow \text{true}')\}$.

Утверждение 23

Для любой аннотированной программы δ на языке HeMo или I-Pascal и формулы ξ тройка $\{AC(\delta, \xi)\} \delta \{ \xi \}$ имеет квазидоказательство, условия корректности которого – это формулы из $VC(\delta, \xi)$ и некоторые пропозициональные тавтологии.

Доказательство проведем индукцией по числу присваиваний и тестов в теле аннотированной программы δ .

База индукции: δ – это некоторое присваивание ($x:=t$) или тест ($\chi?$). В первом случае $AC(\delta, \xi) = \xi_{t/x}$, и тройка $\{AC(\delta, \xi)\} \delta \{ \xi \}$ имеет квазидоказательство, которое просто совпадает с аксиомой для присваивания. Во втором случае $AC(\delta, \xi) = (\chi \rightarrow \xi)$, и тройка $\{AC(\delta, \xi)\} \delta \{ \xi \}$ имеет квазидоказательство, которое просто совпадает с аксиомой для теста.

Предположение индукции: утверждение верно для всех аннотированных программ, состоящих из не более чем $k \geq 1$ присваиваний и тестов.

Шаг индукции: необходимо рассмотреть все варианты, как тело δ «собрано» из более коротких тел, каждое из которых состоит из не более чем k присваиваний и тестов. Возможные варианты для тела δ : $(\alpha ; \beta)$, $(\alpha \cup \beta)$, IF χ THEN α ELSE β , (α^*) и WHILE χ DO α . Но все эти случаи достаточно однотипны, поэтому мы ограничимся разбором только трех случаев $(\alpha ; \beta)$, $(\alpha \cup \beta)$ и WHILE χ DO α .

- Пусть тело δ имеет вид $(\alpha ; \beta)$. В аксиоматической семантике AS или I-PAS имеется правило SC. Так как $AC(\delta, \xi) = AC(\alpha, AC(\beta, \xi))$, то частный случай этого правила имеет следующий вид:

$$\frac{\{AC(\alpha, AC(\beta, \xi))\} \alpha \quad \{AC(\beta, \xi)\} \beta}{\{AC(\delta, \xi)\} \delta \{ \xi \}}$$

Согласно предположению индукции, $\{AC(\alpha, AC(\beta, \xi))\} \alpha$ и $\{AC(\beta, \xi)\} \beta$ имеют квазидоказательства, условия корректности которых – это $VC(\alpha, AC(\beta, \xi))$ и $VC(\beta, \xi)$ соответственно вместе с некоторыми пропозициональными тавтологиями. Но $VC(\delta, \xi) = VC(\alpha, AC(\beta, \xi)) \cup VC(\beta, \xi)$. Следовательно, $\{AC(\delta, \xi)\} \delta \{ \xi \}$ имеет квазидоказа-

тельство, все условия корректности которого – это формулы из $VC(\delta, \xi)$ и некоторые пропозициональные тавтологии.

- Пусть тело δ имеет вид $(\alpha \cup \beta)$. В аксиоматической семантике AS имеются правила NC и PS/CW. Так как $AC(\delta, \xi) = AC(\alpha, \xi) \wedge AC(\beta, \xi)$, то комбинируя эти правила можно построить следующее дерево вывода:

$$\begin{array}{c}
 \frac{(AC(\alpha, \xi) \wedge AC(\beta, \xi)) \rightarrow AC(\alpha, \xi) \quad \{AC(\alpha, \xi)\} \alpha \quad \{\xi\}}{\{AC(\alpha, \xi) \wedge AC(\beta, \xi)\} \alpha \quad \{\xi\}} \quad \frac{(AC(\alpha, \xi) \wedge AC(\beta, \xi)) \rightarrow AC(\beta, \xi) \quad \{AC(\beta, \xi)\} \beta \quad \{\xi\}}{\{AC(\alpha, \xi) \wedge AC(\beta, \xi)\} \beta \quad \{\xi\}} \\
 \hline
 \{AC(\delta, \xi)\} \delta \quad \{\xi\}
 \end{array}$$

Согласно предположению индукции, $\{AC(\alpha, \xi)\} \alpha \quad \{\xi\}$ и $\{AC(\beta, \xi)\} \beta \quad \{\xi\}$ имеют квазидоказательства, условия корректности которых – это некоторые тавтологии и $VC(\alpha, \xi)$ и $VC(\beta, \xi)$ соответственно. Но $VC(\delta, \xi) = VC(\alpha, \xi) \cup VC(\beta, \xi)$, а $(AC(\alpha, \xi) \wedge AC(\beta, \xi)) \rightarrow AC(\alpha, \xi)$ и $(AC(\alpha, \xi) \wedge AC(\beta, \xi)) \rightarrow AC(\beta, \xi)$ – пропозициональные тавтологии. Следовательно, $\{AC(\delta, \xi)\} \delta \quad \{\xi\}$ имеет квазидоказательство, все условия корректности которого – это пропозициональные тавтологии и формулы из $VC(\delta, \xi)$.

- Пусть тело δ имеет вид $WHILE^1 \chi DO \alpha$. В аксиоматической семантике I-PAS имеются правила DL и PS/CW. Так как $AC(\delta, \xi) = 1$, то комбинируя эти правила можно построить следующее дерево вывода:

$$\begin{array}{c}
 (1 \wedge \chi) \rightarrow AC(\alpha, 1) \quad \{AC(\alpha, 1)\} \alpha \quad \{1\} \\
 \hline
 \{1 \wedge \chi\} \alpha \quad \{1\} \\
 \hline
 \{1\} \delta \quad \{1 \wedge \neg \chi\} \quad \quad \quad (1 \wedge \neg \chi) \rightarrow \xi \\
 \hline
 \{AC(\delta, \xi)\} \delta \quad \{\xi\}
 \end{array}$$

Согласно предположению индукции, $\{AC(\alpha, 1)\} \alpha \quad \{1\}$ имеет квазидоказательство, условия корректности которых – это $VC(\alpha, 1)$ вместе с некоторыми пропозициональными тавтологиями. Но $VC(\delta, \xi) = VC(\alpha, 1) \cup \{((1 \wedge \neg \chi) \rightarrow \xi), ((1 \wedge \chi) \rightarrow AC(\alpha, 1))\}$ Следовательно, $\{AC(\delta, \xi)\} \delta \quad \{\xi\}$

имеет квазидоказательство, все условия корректности которого – это формулы из $VC(\delta, \xi)$ и некоторые пропозициональные тавтологии.

На этом мы считаем доказательство шага индукции завершённым. ■

Из доказанного утверждения следует, что любая аннотированная тройка $\{\phi\} \alpha \{\psi\}$ (на языке HeMo или I-Pascal) имеет квазидоказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\alpha, \psi)$ и формула $(\psi \rightarrow AC(\alpha, \psi))$. Оказывается, что если тройка правильная, то это квазидоказательство является доказательством.

Утверждение 24

Для любой аннотированной тройки на языке HeMo или I-Pascal $\{\chi\} \delta \{\xi\}$ всякое доказательство ее правильности может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\alpha, \psi)$ и формула $(\phi \rightarrow AC(\alpha, \psi))$.

Доказательство (набросок) проведем индукция по высоте доказательства правильности троек.

База индукции: доказательство имеет высоту 0, то есть $\{\chi\} \delta \{\xi\}$ является аксиомой. Это возможно только в том случае, когда δ – это или некоторое присваивание или некоторый тест.

- В случае присваивания наша тройка имеет вид $\{\xi_{t/x}\} (x := t) \{\xi\}$ и, следовательно, $\chi \equiv \xi_{t/x}$. В тоже время, $AC((x := t), \xi) = \xi_{t/x}$, и, следовательно, $(\chi \rightarrow AC(\delta, \xi))$ – истинная формула. Поэтому квазидоказательство тройки $\{AC(\delta, \xi)\} \delta \{\xi\}$ является доказательством.
- В случае теста наша тройка имеет вид $\{\theta \rightarrow \xi\} (\theta?) \{\xi\}$ и, следовательно, $\chi \equiv (\theta \rightarrow \xi)$. В тоже время, $AC((\theta?), \xi) = (\theta \rightarrow \xi)$, и, следовательно, $(\chi \rightarrow AC(\delta, \xi))$ – истинная формула. Поэтому квазидоказательство тройки $\{AC(\delta, \xi)\} \delta \{\xi\}$ является доказательством.

Предположение индукции: утверждение верно для всех правильных троек, которые имеют доказательство правильности высоты не более $k \geq 0$.

Шаг индукции: пусть тройка $\{\chi\} \delta \{\xi\}$ имеет доказательство правильности высоты $(k+1)$. Так как $(k+1) \geq 1$, то это доказательство не является аксио-

мой, а начинается с применения некоторого правила. Поэтому нам следует разобрать все возможные случаи, какое именно это правило. Мы, однако, ограничимся разбором только двух из четырех правил аксиоматической семантики AS языка HeMo – правил LI и NC. Разбор правил PS/WC и SC требует доказательства одной технической леммы, а разбор правил аксиоматической семантики I-PAS языка I-Pascal проходит аналогично правилам AS.

- Если доказательство правильности тройки $\{\chi\}\delta\{\xi\}$ начинается с правила для недетерминированного цикла, то δ имеет вид α^* , $\chi \equiv \xi \equiv \iota$, а тройка $\{\iota\}\alpha\{\iota\}$ имеет доказательство правильности высоты не более k . В тоже время $AC(\alpha^*, \iota) = \iota$, и, следовательно, $(\chi \rightarrow AC(\delta, \xi))$ – истинная формула. По предположению индукции доказательство правильности тройки $\{\iota\}\alpha\{\iota\}$ может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\alpha, \iota)$ и формула $(\iota \rightarrow AC(\alpha, \iota))$. Но $VC(\alpha^*, \iota) = VC(\alpha, \iota) \cup \{(\iota \rightarrow \iota)\}$. Следовательно, выбранное доказательство правильности тройки $\{\chi\}\delta\{\xi\}$ может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\delta, \xi)$ и формула $(\chi \rightarrow AC(\delta, \xi))$.
- Если доказательство правильности тройки $\{\chi\}\delta\{\xi\}$ начинается с правила для недетерминированного выбора, то δ имеет вид $(\alpha \cup \beta)$, а тройки $\{\chi\}\alpha\{\xi\}$ и $\{\chi\}\beta\{\xi\}$ имеют доказательства правильности высоты не более k . По предположению индукции первое из этих доказательств может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\alpha, \xi)$ и формула $(\chi \rightarrow AC(\alpha, \xi))$. Аналогично, второе из этих доказательств может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\beta, \xi)$ и формула $(\chi \rightarrow AC(\beta, \xi))$. В тоже время $AC((\alpha \cup \beta), \xi) = AC(\alpha, \xi) \wedge AC(\beta, \xi)$; следовательно, $(\chi \rightarrow AC(\delta, \xi))$ – истинная формула. Далее, $AC((\alpha \cup \beta), \xi) = AC(\alpha, \xi) \wedge AC(\beta, \xi)$; следовательно, выбранное доказательство правильности тройки $\{\chi\}\delta\{\xi\}$ может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии, формулы множества $VC(\delta, \xi)$ и формула $(\chi \rightarrow AC(\delta, \xi))$.

На этом мы завершаем набросок доказательства шага индукции. ■

Верификация программ

Из Утверждение 23 и Утверждение 24 получаем

Утверждение 25

Для любой аннотированной тройки Хоара на языке НеМо или I-Pascal $\{\varphi\} \alpha \{\psi\}$ множество утверждений $\{(\varphi \rightarrow AC(\alpha, \psi))\} \cup VC(\alpha, \psi)$ является ее условиями корректности.

Доказательство. Пусть $VC = \{(\varphi \rightarrow AC(\alpha, \psi))\} \cup VC(\alpha, \psi)$. Надо показать, что формулы множества VC истинны тогда и только тогда, когда $\{\varphi\} \alpha \{\psi\}$ – правильная тройка.

Если все формулы из VC истинны, то (в силу Утверждение 23) аннотированная тройка $\{AC(\alpha, \psi)\} \alpha \{\psi\}$ имеет доказательство, в котором все условия корректности – это пропозициональны тавтологии и формулы из $VC(\alpha, \psi)$. Следовательно, в аксиоматической семантике (AS или I-PAS) можно доказать правильность тройки $\{\varphi\} \alpha \{\psi\}$, если начать с применения правила PS/CW:

$$(\varphi \rightarrow AC(\alpha, \psi)) \quad \{AC(\alpha, \psi)\} \alpha \{\psi\}$$

$$\{\varphi\} \alpha \{\psi\}$$

Если же $\{\varphi\} \alpha \{\psi\}$ имеет доказательство правильности, то это доказательство (в силу Утверждение 24) может быть перестроено в доказательство, в котором все условия корректности – это пропозициональные тавтологии и формулы множества VC . ■

В силу Утверждение 22, для любой аннотированной тройки $\{\varphi\} \alpha \{\psi\}$ на языке НеМо или I-Pascal правильность влечет истинность. Доказанное Утверждение 25 означает, что у нас есть надежный и полный алгоритм генерации условий корректности для доказательства правильности (верификации) аннотированных троек на языках НеМо и I-Pascal: для этого необходимо и достаточно сгенерировать множество условий корректности $VC\{\alpha, \psi\} \cup \{(\varphi \rightarrow AC(\alpha, \psi))\}$, а затем проверить (или доказать) истинность всех получившихся формул. Но вот здесь-то и лежит камень преткновения...

Дело в том, что у нас не такой уж большой выбор средств проверки и/или доказательства истинности формул. В нашем курсе мы познакомились только с методами верификации пропозициональных формул (см. лекцию 6 и лабораторную работу 1). Мы также рекомендовали для более глу-

бокого знакомства с методами поиска доказательства монографию¹⁴⁸ и дополнительно можем рекомендовать краткий обзор по разрешимым теориям¹⁴⁹. Однако можно утверждать, что эффективные методы поиска доказательства или разрешающие процедуры существуют только для весьма ограниченных классов формул.

Получается несколько неутешительный вывод: мы научились эффективно сводить задачу верификации аннотированных вычислительных программ (на модельных языках HeMo и I-Pascal) к задаче верификации множеств формул, которая имеет алгоритмическое решение только в очень специальных частных случаях (пропозициональном, например). Отсюда можно прийти к еще более пессимистическому «заключению» о невозможности положительного ответа на «большой вызов» Антони Хоара: сколько-нибудь мощный и практичный верифицирующий транслятор невозможен...

Однако, дело обстоит далеко не так плохо.

Во-первых, полностью автоматическая верификация, возможная для специальных классов программ, может иметь огромное практическое значение. – Так, в частности, для программ, моделирующих устройства с конечным числом возможных состояний, автоматическая верификация стала мощным инструментом обеспечения корректности, надежности и безопасности¹⁵⁰. Формальным выражением признания достигнутых результатов в этом направлении верификации программ стало присуждение Ассоциацией Машинных Вычислений в 2007 г. премии имени Алана М. Тьюринга «за выдающийся вклад в развитие высокоэффективного метода верификации моделей, широко используемого для верификации программного и аппаратного обеспечения» американским ученым Эдмунду Кларку и Алану Эмерсону и французскому ученому Джозефу Сифакису.

И, во-вторых, за 50 лет после возникновения верификации программ как самостоятельного направления исследований, были разработаны не только специализированные алгоритмы поиска доказательства или проверки истинности, но и методы комбинирования разных алгоритмов поиска доказательства и проверки истинности. Наиболее известны, видимо, исследования, ведущиеся в лаборатории Информатики и Программирования¹⁵¹ в Станфорде, США, в независимом некоммерческом институте SRI

¹⁴⁸ Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. – М.: Наука, 1983.

¹⁴⁹ Рабин М.О. *Разрешимые теории*. В кн.: Справочная книга по математической логике, ч.3. Теория рекурсии. – М.: Наука, 1982. с. 77-111.

¹⁵⁰ Смотрите, например, монографию Кларк Э., М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М: МЦНМО, 2002.

¹⁵¹ Computer Science Laboratory, URL <http://www.csl.sri.com/index.html>.

International¹⁵² под руководством Дж. Рушби (J. Rushby). Особенность подхода, принятого в SRI International, состоит в том, что исследования ведутся в русле автоматического доказательства теорем, в формулировках которых участвуют понятия из разных теорий первого порядка. Для первоначального знакомства с этими исследованиями в целом можно рекомендовать обзорную статью самого Дж. Рушби Theorem Proving for Verification¹⁵³.

Следует обратить внимание на то, что актуальность проблемы создания верифицирующего транслятора осознается не только академической и инженерной общественностью, но и на государственном уровне во многих странах. Так, например, в Федеративной Республике Германии уже на протяжении 6 лет развивается национальный проект верифицирующего компилятора Verisoft¹⁵⁴. Финансирование проекта осуществляется через Министерство образования и науки (BMBF), управление – через Германский аэрокосмический центр (DLR). Бюджет на 2007-2010 гг. составляет 12 миллионов Евро.

В то же время в России ситуация с исследованиями по верификации выглядит, на наш взгляд, значительно хуже. Так, последняя (и, возможно, единственная) монография¹⁵⁵ российских ученых по верификации была выпущена еще в СССР в 1988 г. (Она не потеряла своего научного значения благодаря оригинальным научным результатам, в ней представленным.) Поэтому хочется пожелать, что бы и в Российской Федерации осознание важности исследований в направлении создания верифицирующего транслятора было осознано и поддержано должным образом на должном уровне.

¹⁵² URL <http://www.sri.com/index.html>, бывший в 1946-1977 гг. Stanford Research Institute.

¹⁵³ Rushby J. Theorem Proving for Verification. Lecture Notes in Computer Science, v. 2067, p.39-56, 2001. Статья доступна на URL <http://www.csl.sri.com/papers/movep2k>.

¹⁵⁴ URL <http://www.verisoft.de>.

¹⁵⁵ Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. – М.: Радио и связь, 1988.