

Оглавление

1	Введение	9
1.1	Проблема верификации программ	9
1.2	Необходимые математические понятия	11
1.2.1	Термы и связанные с ними понятия	11
1.2.2	Примеры типов и функциональных символов	13
1.2.3	Подстановки	15
I	Последовательные нерекурсивные программы	17
2	Программы, представленные в виде блок-схем	19
2.1	Понятие блок-схемы	19
2.2	Выполнение блок-схемы	21
2.3	Массивы	22
2.4	Примеры блок-схем	22
2.5	Задача верификации блок-схем	24
3	Метод инвариантов для верификации блок-схем	27
3.1	Базовые множества и базовые пути	27
3.2	Описание метода инвариантов для верификации блок-схем	28
3.3	Обоснование метода инвариантов	29
3.4	Примеры фундированных множеств	30
3.5	Применение метода инвариантов	30
3.5.1	Верификация блок-схемы вычисления суммы	31
3.5.2	Верификация блок-схемы деления с остатком	32
3.5.3	Верификация блок-схемы извлечения корня	33
3.5.4	Верификация блок-схемы возведения в степень	34
3.5.5	Верификация блок-схемы сортировки	35
4	Процессные представления блок-схем	39
4.1	Понятие процесса	39
4.1.1	Действия	39
4.1.2	Процессы и их выполнение	40
4.2	Процессы, соответствующие блок-схемам	41
4.3	Верификация блок-схем с использованием процессного представления	42

5	Верификация операторных программ	45
5.1	Понятие операторной программы	45
5.2	Примеры операторных программ	46
5.3	Метод инвариантов для верификации операторных программ	47
5.4	Пример верификации операторной программы	48
6	Задачи	49
6.1	Задачи без массивов	49
6.1.1	Произведение двух чисел	49
6.1.2	Возведение в степень	49
6.1.3	Извлечение квадратного корня	50
6.1.4	Извлечение логарифма	50
6.1.5	Вычисление частного и остатка от деления целых чисел	51
6.1.6	Наибольший общий делитель	51
6.1.7	Представление наибольшего общего делителя линейной формой	53
6.1.8	Наибольший общий делитель и наименьшее общее кратное	54
6.1.9	Приближенное решение уравнения	54
6.1.10	Проверка на простоту	54
6.1.11	Проверка, является ли число совершенным	55
6.2	Задачи с массивами	55
6.2.1	Инвертирование массива	55
6.2.2	Минимальный элемент массива	56
6.2.3	Двоичный поиск	56
6.2.4	Наибольший общий делитель компонентов массива	57
6.2.5	Список простых чисел от 2 до n	57
6.2.6	Сортировка массива	57
6.2.7	Перестановка массива с заданным условием	58
6.2.8	Перестановка массива в заданном порядке	59
6.2.9	Вычисление определителя матрицы	59
6.3	Другие задачи	60
6.4	Исследовательские проблемы	61
II	Верификация функциональных программ	63
7	Введение в функциональное программирование	65
7.1	Парадигма функционального программирования	65
7.2	Примеры функциональных программ	66
7.2.1	Конкатенация строк	66
7.2.2	Инвертирование строки	67
7.2.3	Поиск подстроки	67
8	Функциональные программы	69
8.1	Пополненные домены и функции на них	69
8.1.1	Пополненные домены	69
8.1.2	Монотонные функции	69
8.1.3	Естественные продолжения	70

8.1.4	Частично упорядоченные множества монотонных функций	71
8.1.5	Полные частично упорядоченные множества	71
8.2	Функциональные программы	72
8.2.1	Понятие функциональной программы	72
8.2.2	Функционал, соответствующий функциональной программе	72
8.2.3	Непрерывные функционалы на полных частично упорядоченных множествах	73
8.2.4	Неподвижные точки функционалов на полных частично упорядоченных множествах	73
8.2.5	Непрерывность функционалов, соответствующих функциональным программам	74
8.2.6	Нахождение наименьших неподвижных точек функциональных программ	77
8.2.7	Примеры неподвижных точек функциональных программ	77
8.2.8	Немонотонные функциональные программы	78
8.3	Алгоритмическая полнота функциональных программ	79
9	Вычисление значений наименьших неподвижных точек	81
9.1	Постановка задачи	81
9.2	Метод решения	81
9.3	Вычислительные правила	82
9.4	Упрощение термина	83
9.5	Функция C_{Σ}	87
9.6	Вспомогательные понятия	89
9.6.1	Полные раскрытия	89
9.6.2	Индексированные термины	90
9.6.3	Σ -переходы	90
9.7	Безопасные вычислительные правила	92
9.8	Свойства правил PO, LO, PI, LI	97
9.8.1	Безопасность правила PO	97
9.8.2	Безопасность правила LO	99
9.8.3	Пример небезопасности правила LO	101
9.8.4	Пример небезопасности правил PI и LI	101
10	Верификация функциональных программ	103
10.1	Задача верификации функциональных программ	103
10.2	Метод вычислительной индукции	103
10.2.1	Описание метода	103
10.2.2	Примеры верификации функциональных программ методом вычислительной индукции	104
10.3	Метод структурной индукции	108
10.3.1	Описание метода	108
10.3.2	Примеры верификации функциональных программ методом структурной индукции	109
10.4	Другие методы верификации функциональных программ	114

10.4.1	Оценка наименьшей неподвижной точки функциональной программы сверху	114
10.4.2	Эквивалентные преобразования функциональных программ	115
11	Задачи	119
11.1	Нахождение наименьших неподвижных точек функциональных программ	119
11.2	Доказательство того, что наименьшая неподвижная точка имеет заданный вид	119
11.3	Совпадение функций, определяемых функциональными программами . .	120
11.4	Свойства наименьших неподвижных точек функциональных программ .	123
11.5	Исследовательские проблемы	126
III	Model checking	127
12	Модели программных систем	129
12.1	Верификация программных систем	129
12.1.1	Математические модели систем	129
12.1.2	Спецификация	130
12.1.3	Построение формальных доказательств	130
12.2	Системы переходов	131
12.2.1	Понятие системы переходов	131
12.2.2	Пути в системах переходов	132
12.2.3	Построение системы переходов, соответствующей программной системе	132
13	Model checking на основе CTL	135
13.1	Темпоральная логика CTL	135
13.1.1	Формулы темпоральной логики CTL	135
13.1.2	Значения CTL-формул	136
13.1.3	Эквивалентность CTL-формул	136
13.1.4	Примеры свойств программных систем, выражаемых CTL-формулами	138
13.2	Задача model checking для CTL	138
13.3	MC-CTL на основе понятия неподвижной точки	140
13.3.1	Неподвижные точки монотонных операторов	140
13.3.2	Вычисление множеств $(\mathbf{EU}(B, C))^S$ и $(\mathbf{EGB})^S$ на основе понятия неподвижной точки	140
13.3.3	Алгоритм решения задачи MC-CTL на основе понятия неподвижной точки	142
13.4	μ -исчисление	143
13.4.1	μ -формулы	143
13.4.2	Значения μ -формул	144
13.4.3	Ускоренное вычисление значений μ -формул	147
13.4.4	Вложение CTL в μ -исчисление	148

14 Бинарные диаграммы решений	149
14.1 Бинарные диаграммы решений и связанные с ними понятия	149
14.1.1 Определение бинарной диаграммы решений	149
14.1.2 Эквивалентность и изоморфность бинарных диаграмм решений	149
14.1.3 Представление множеств замкнутых подстановок бинарными диаграммами решений	150
14.1.4 Редукция бинарных диаграмм решений	151
14.1.5 Представление булевых функций	151
14.1.6 Подстановка значений вместо переменных	152
14.2 Согласованность с порядком переменных	152
14.3 Алгебраические операции на бинарных диаграммах решений	155
14.3.1 Булевы операции	155
14.3.2 Произведение	156
14.4 MC-CTL с использованием бинарных диаграмм решений	158
14.5 Оптимизирующие преобразования	159
15 Model checking на основе LTL	161
15.1 Формулы темпоральной логики LTL	161
15.2 Квантифицированные LTL-формулы	162
15.3 Задачи model checking для LTL	163
15.3.1 Система переходов Σ_A	163
15.3.2 Система переходов $\Sigma \times \Sigma_A$	164
15.3.3 Первая задача model checking для LTL	166
15.3.4 Вторая задача model checking для LTL	167
15.4 Автоматы Бюхи	169
15.4.1 Понятие автомата Бюхи	169
15.4.2 Язык автомата	170
15.4.3 Эквивалентность автоматов	172
15.4.4 Пересечение автоматов	172
15.4.5 Использование автоматов Бюхи для MC-LTL	173
15.4.6 Оптимизация построения автомата \mathcal{B}_A	173
16 Вероятностный model checking	177
16.1 Введение	177
16.2 Вероятностные системы переходов	178
16.2.1 Понятие вероятностной системы переходов	178
16.2.2 Примеры вероятностных систем переходов	179
16.3 Темпоральная логика PCTL	180
16.3.1 Свойства вероятностных систем переходов	180
16.3.2 Формулы логики PCTL	181
16.3.3 Значения формул логики PCTL в состояниях вероятностных систем переходов	181
16.3.4 Интерпретация значений формул логики PCTL	183
17 Исследовательские проблемы	185

Глава 1

Введение

1.1 Проблема верификации программ

Проблема верификации (т.е. доказательства правильности) программ занимает центральное положение в теории и практике разработки программного обеспечения. Под правильностью программ понимается их соответствие различным условиям корректности, безопасности, устойчивости в случае непредусмотренного поведения окружения, эффективности использования ресурсов времени и памяти, оптимальности реализованных в программе алгоритмов, и т.п.

Как правило, для обоснования правильности программы её тестируют, т.е. анализируют её поведение на некоторых входных данных. Однако тестирование обладает очевидным недостатком: если его возможно провести не для всех допустимых входных данных, а только лишь для их небольшой части (что имеет место почти всегда), то оно не может служить гарантированным обоснованием того, что тестируемая программа обладает проверяемыми свойствами. Как отметил один из основоположников программирования Э.В.Дейкстра [1], «тестирование может лишь помочь выявить некоторые ошибки, но отнюдь не доказать их отсутствие».

Ошибки в программах могут быть весьма тонкими, но во многих программах наличие даже незначительных ошибок категорически недопустимо. Например, наличие ошибок в таких программах, как

- программы управления атомными электростанциями,
- программы, управляющие работой медицинских устройств,
- программы в бортовых системах управления самолетов и космических аппаратов,
- программы в системах управления секретными базами данных, системах электронной коммерции, и т.п.

может привести к существенному ущербу для экономики и жизни людей.

Приведем один пример, иллюстрирующий наличие ошибок даже в очень простых программах, правильность которых на первый взгляд не вызывает никакого сомнения. Рассмотрим программу P , задача которой заключается в зачислении денег на счет клиента банка. Количество денег на счету этого клиента хранится в базе данных банка

в переменной x . Когда P выполняет действия по зачислению суммы s на этот счет, она выполняет следующие действия:

- копирует в свою внутреннюю память значение переменной x
- вычисляет новое значение, которое должна иметь переменная x , оно равно сумме текущего значения x и зачисляемой суммы s , и
- заносит в переменную x это новое значение.

Даже если программа P выполняет все свои действия правильно, это не гарантирует корректности обслуживания клиента в том случае, когда состояние его счета может изменяться несколькими такими программами. Рассмотрим ситуацию, когда состояние счета клиента изменяют две программы P_1 и P_2 описанного выше типа. Возможен следующий вариант совместного выполнения этих программ:

- сначала программа P_1 выполняет свое первое действие, оно начинается в момент времени t_1 , а заключительное действие P_1 (обновление значения x) происходит в момент времени t_2 , и
- в момент времени, лежащий в интервале между t_1 и t_2 , программа P_2 начинает свою операцию зачисления денег на счет клиента, причем выполнение первого действия программы P_2 (копирование значения переменной x) производится до выполнения заключительного действия программы P_1 .

После завершения работы обеих программ те деньги, которые зачислила на счет клиента одна из программ P_1 , P_2 , просто пропадут.

Ошибку описанного выше типа можно не обнаружить путем тестирования, т.к. операция зачисления денег на счет клиента выполняется практически мгновенно, и поэтому среди тестов, которыми можно анализировать программы подобного типа, с большой вероятностью могут отсутствовать такие тесты, в которых две различные программы, обслуживающие счета клиентов, почти одновременно обращаются к одному и тому же ресурсу памяти.

Если же в результате какого-либо тестирования указанная выше ошибка обнаруживается, и для её исправления конструируются специальные программные механизмы (семафоры, и т.п.) с целью задания правильной дисциплины обращения программ к одному и тому же ресурсу памяти, то встает вопрос о том, насколько эти механизмы соответствуют своему предназначению (в частности, защищены ли семафоры от непредусмотренного и неавторизованного изменения их значений). Это тоже может анализироваться путем тестирования, и опять может получиться так, что среди тестов, которыми анализируется поведение указанных выше специальных программных механизмов, с большой вероятностью будут отсутствовать такие тесты, в которых проявляется некорректное поведение этих механизмов.

Гарантированное обоснование правильности программ может быть получено только при помощи альтернативного подхода, принципиально отличного от тестирования. Данный подход называется **верификацией**. В самом общем виде верификация программы может пониматься как построение математического доказательства утверждения о том, что верифицируемая программа соответствует своему предназначению. Предназначение программы может быть выражено, например, путем описания функции, которую

должна вычислять эта программа, или правил взаимодействия этой программы с другими программами, т.е. реакции, которую эта программа должна обеспечивать в ответ на получение сигналов или сообщений от других программ.

Формальное описание предназначения программы (или некоторых свойств, которыми она должна обладать) в виде математического утверждения называется **спецификацией** этой программы. Спецификация может представлять собой формальное описание самых разнообразных свойств программы, например:

- её входные и выходные данные находятся в заданном соотношении,
- программа всегда завершает свою работу,
- во время работы программы не происходит сбоев и ненормальных ситуаций (например, деления на 0, извлечения квадратного корня из отрицательного числа, выхода индекса за границы массива, неавторизованных утечек информации, и т.п.),
- программа решает свою задачу за установленное время, и использует не более установленного объема памяти.

Для верификации программы P необходимо определить

- математический смысл всех конструкций, используемых в P , называемый **формальной семантикой** (или просто **семантикой**) этих конструкций, и
- спецификацию $Spec$ этой программы, выражающую то свойство программы P , которое необходимо верифицировать,

после чего можно ставить вопрос о верификации P относительно $Spec$, т.е. о построении математического доказательства утверждения о том, что P удовлетворяет $Spec$.

1.2 Необходимые математические понятия

1.2.1 Термы и связанные с ними понятия

Мы будем предполагать, что заданы следующие множества.

- Множество $Types$, элементы которого называются **типами**. Мы будем понимать типы так же, как понимаются типы данных в языках программирования. Каждому типу $\tau \in Types$ сопоставлено множество D_τ **значений** типа τ , называемое **доменом** типа τ .
- Множество Var , элементы которого называются **переменными**. Каждой переменной $x \in Var$ сопоставлен тип $\tau(x) \in Types$. Каждая переменная $x \in Var$ может принимать **значения** в домене $D_{\tau(x)}$, т.е. в различные моменты времени переменная x может быть связана с различными элементами домена $D_{\tau(x)}$.

- Множество Con , элементы которого называются **константами**. Каждой константе $c \in Con$ сопоставлены тип $\tau(c) \in Types$ и значение из $D_{\tau(c)}$, обозначаемое тем же символом c , и называемое **интерпретацией** константы c . Будем считать, что $\forall \tau \in Types$ любой элемент $d \in D_\tau$ является константой типа τ , которой соответствует сам элемент d .
- Множество Fun , элементы которого называются **функциональными символами (ФС)**. Каждому ФС $f \in Fun$ сопоставлены

– **функциональный тип (ФТ)** $\tau(f)$, который представляет собой запись вида

$$(\tau_1, \dots, \tau_n) \rightarrow \tau, \quad (1.1)$$

где $\tau_1, \dots, \tau_n, \tau \in Types$, и

- частичная функция вида $D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_\tau$, где $\tau(f)$ имеет вид (1.1), данная функция обозначается тем же символом f и называется **интерпретацией** ФС f .

(напомним, что функция $f : A \rightarrow B$ называется **частичной**, если $\forall a \in A$ значение $f(a)$ м.б. не определено)

Функциональной переменной называется переменная, тип которой является функциональным типом. Множество всех функциональных переменных обозначается записью $FVar$.

Термы строятся из переменных, констант и ФС. Множество всех термов обозначается символом Tm . Каждый терм e имеет тип $\tau(e) \in Types$, определяемый структурой терма e .

Правила построения термов имеют следующий вид:

- каждая переменная и константа является термом того типа, который сопоставлен этой переменной или константе, и
- если $e_1, \dots, e_n \in Tm$, $f \in Fun \cup FVar$, и $\tau(f)$ имеет вид (1.1), где $\tau_1 = \tau(e_1), \dots, \tau_n = \tau(e_n)$, то запись $f(e_1, \dots, e_n)$ – терм типа τ .

Терм $e \in Tm$ называется **подтермом** терма $e' \in Tm$, если либо $e = e'$, либо $e' = f(e_1, \dots, e_n)$, где $f \in Fun \cup FVar$, и $\exists i \in \{1, \dots, n\}$: e – подтерм терма e_i . Запись $e \subseteq e'$, где $e, e' \in Tm$, означает, что e – подтерм e' . Запись $e \subset e'$, где $e, e' \in Tm$, означает, что $e \subseteq e'$ и $e \neq e'$.

Индукцией по структуре терма $e \in Tm$ нетрудно доказать, что

$$\begin{aligned} &\text{если } e_1 \text{ и } e_2 \text{ – различные подтермы терма } e, \text{ то либо } e_1 \subset e_2, \\ &\text{либо } e_2 \subset e_1, \text{ либо } e_1 \text{ и } e_2 \text{ не имеют общих компонентов.} \end{aligned} \quad (1.2)$$

Запись $x \in e$, где $x \in Var$, $e \in Tm$ означает, что x входит в e .

Будем использовать следующие обозначения и соглашения:

- $\forall e \in Tm$ $Var(e)$ обозначает множество $\{x \in Var \mid x \in e\}$,
- $\forall e_1, \dots, e_n \in Tm$ $Var(e_1, \dots, e_n) = Var(e_1) \cup \dots \cup Var(e_n)$,

- $\forall X \subseteq Var\ Tm(X)$ обозначает множество $\{e \in Tm \mid Var(e) \subseteq X\}$,
- $FVar(e)$ обозначает множество $Var(e) \cap FVar$,
- $\forall \tau \in Types$ запись Tm_τ обозначает множество $\{e \in Tm \mid \tau(e) = \tau\}$, $\forall E \subseteq Tm$ запись E_τ обозначает множество $E \cap Tm_\tau$, запись E_X обозначает множество $E \cap Var$,
- в терме вида $f(e)$ скобки слева и справа от e могут опускаться, если $\tau(f)$ имеет вид $(\tau) \rightarrow \tau'$,
- для каждой рассматриваемой функции $f : E \rightarrow E'$, где $E, E' \subseteq Tm$, будем предполагать, что $\forall e \in E \ \tau(e) = \tau(f(e))$.

Терм $e \in Tm$ **замкнут**, если $Var(e) = \emptyset$. Каждому замкнутому терму e соответствует объект $eval(e)$, называемый **значением** данному терма, и либо является элементом $D_{\tau(e)}$, либо не определён. Если e – константа, то $eval(e)$ – интерпретация этой константы, и если e имеет вид $f(e_1, \dots, e_n)$, то $eval(e)$ определён только если определено значение функции f на кортеже $(eval(e_1), \dots, eval(e_n))$, и в этом случае $eval(e)$ равен этому значению. Для каждого замкнутого терма e будем обозначать объект $eval(e)$ той же записью, что и сам терм (т.е. e).

1.2.2 Примеры типов и функциональных символов

Арифметические типы и функциональные символы

Con содержит типы **N** и **I**, значениями которых являются натуральные $(0, 1, \dots)$, и целые числа, соответственно.

Fun содержит ФС $+$, $-$, \cdot , div , mod типа $(\mathbf{I}, \mathbf{I}) \rightarrow \mathbf{I}$, и

- функции $+$, $-$, и \cdot представляют собой соответствующие арифметические операции,
- div – частичная функция (определённая только когда второй аргумент отличен от 0), mod – частичная функция (определённая только когда второй аргумент больше 0), div и mod вычисляют частное и остаток соответственно от деления первого аргумента на второй.

Термы $+(e_1, e_2)$, $-(e_1, e_2)$, $\cdot(e_1, e_2)$, $div(e_1, e_2)$, $mod(e_1, e_2)$ будут записываться в виде $e_1 + e_2$, $e_1 - e_2$, $e_1 e_2$, e_1 / e_2 и $e_1 \% e_2$ соответственно.

Логические типы и функциональные символы

Types содержит тип **B**, и $D_B = \{0, 1\}$. Термы типа **B** называются **формулами**. Множество всех формул обозначается Fm . $\forall X \subseteq Var$ запись $Fm(X)$ обозначает множество $Tm(X) \cap Fm$. При построении формул могут использоваться обычные булевские ФС (\neg , \wedge , \vee , \rightarrow и т.д.), которым соответствуют функции отрицания, конъюнкции, дизъюнкции, и т.д. Символ 1 обозначает тождественно истинную формулу, а символ 0 – тождественно ложную формулу. Формулы вида $\wedge(e_1, e_2)$, $\vee(e_1, e_2)$, и т.п. мы будем записывать в

более привычном виде $e_1 \wedge e_2$, $e_1 \vee e_2$, и т.д. Формулы вида $e_1 \wedge \dots \wedge e_n$ и $e_1 \vee \dots \vee e_n$ могут также записываться в виде $\left\{ \begin{matrix} e_1 \\ \vdots \\ e_n \end{matrix} \right\}$ и $\left[\begin{matrix} e_1 \\ \vdots \\ e_n \end{matrix} \right]$ соответственно, а также в виде $\bigwedge_{i \in \{1, \dots, n\}} e_i$ и $\bigvee_{i \in \{1, \dots, n\}} e_i$ соответственно. Формулы вида $\neg e$ могут обозначаться \bar{e} .

Разные ФС могут иметь одинаковое обозначение. Ниже мы приводим примеры таких ФС. Для каждого типа τ

- *Fun* содержит ФС *eq* типа $(\tau, \tau) \rightarrow \mathbf{B}$, которому соответствует функция отображающая пару $(d_1, d_2) \in D_\tau \times D_\tau$ в элемент 1, если $d_1 = d_2$, и 0, если $d_1 \neq d_2$,
- если на D_τ задано отношение частичного порядка, то *Fun* содержит ФС $<, \leq, >, \geq$ типа $(\tau, \tau) \rightarrow \mathbf{B}$. Каждому из этих ФС соответствует функция отображающая каждую пару $(d_1, d_2) \in D_\tau \times D_\tau$ в элемент
 - 1, если $d_1 < d_2, d_1 \leq d_2, d_1 > d_2, d_1 \geq d_2$ соответственно, и
 - 0, если соответствующее соотношение неверно,
- *Fun* содержит ФС *if_then_else* типа $(\mathbf{B}, \tau, \tau) \rightarrow \tau$. Соответствующая функция отображает тройку вида $(1, d_1, d_2)$ в d_1 , и тройку вида $(0, d_1, d_2)$ в d_2 .

Термы *eq*(e_1, e_2), $<$ (e_1, e_2), и т.д. будут записываться в виде $e_1 = e_2$, $e_1 < e_2$, и т.д., соответственно. Термы *if_then_else*(e, e_1, e_2) будут записываться в виде *if e then e₁ else e₂*, или $e?e_1 : e_2$.

Строковые типы и функциональные символы

Types содержит типы **C** и **S**. значения которых называются **символами** и **символьными строками** (или просто **строками**) соответственно. Каждая строка представляет собой последовательность символов $a_1 \dots a_n$, где $n \geq 0$. При $n = 0$ эта последовательность пустая и обозначается ε .

Fun содержит

- ФС *head* и *tail* типа **S** \rightarrow **C** и **S** \rightarrow **S** соответственно, которым соответствуют частичные функции, определенные только для непустых строк, данные функции отображают строку $a_1 \dots a_n$ в символ a_1 и строку $a_2 \dots a_n$ соответственно (называемые **головой** и **хвостом** строки $a_1 \dots a_n$ соответственно),
- ФС *conc* типа $(\mathbf{C}, \mathbf{S}) \rightarrow \mathbf{S}$, которому соответствует функция, отображающая пару $(a, a_1 \dots a_n)$ в строку $aa_1 \dots a_n$.

Термы *conc*(e, e'), *head*(e), *tail*(e) будем записывать в сокращенном виде ee' , e_h и e_t , соответственно.

Кортежные типы и функциональные символы

Для каждого списка типов τ_1, \dots, τ_n (некоторые компоненты этого списка могут совпадать)

- *Types* содержит тип, обозначаемый записью (τ_1, \dots, τ_n) , и

$$D_{(\tau_1, \dots, \tau_n)} = D_{\tau_1} \times \dots \times D_{\tau_n},$$

- *Fun* содержит ФС *tuple* типа $(\tau_1, \dots, \tau_n) \rightarrow (\tau_1, \dots, \tau_n)$, которому соответствует тождественная функция, термы вида $tuple(e_1, \dots, e_n)$ будут обозначаться записью (e_1, \dots, e_n) .

1.2.3 Подстановки

Подстановкой называется функция $\theta : Var \rightarrow Tm$. Будем говорить, что подстановка θ заменяет переменную $x \in Var$ на терм $\theta(x)$.

Будем использовать следующие обозначения:

- множество всех подстановок обозначается символом Θ ,
- $\forall \theta \in \Theta$ запись $Var(\theta)$ обозначает множество $\{x \in Var \mid \theta(x) \neq x\}$,
- $\forall X \subseteq Var \quad \Theta(X) = \{\theta \in \Theta \mid Var(\theta) \subseteq X\}$,
- подстановка $\theta \in \Theta$ может обозначаться записями

$$x \mapsto \theta(x) \quad \text{или} \quad (\theta(x_1)/x_1, \dots, \theta(x_n)/x_n), \quad (1.3)$$

вторая запись в (1.3) используется, когда $Var(\theta) = \{x_1, \dots, x_n\}$,

- $\forall \theta \in \Theta, \forall e \in Tm$ запись e^θ обозначает терм, получаемый из e заменой $\forall x \in Var(e)$ каждого вхождения x в e на терм $\theta(x)$,
- $\forall \theta, \theta' \in \Theta$ запись $\theta\theta'$ обозначает подстановку $x \mapsto (x^\theta)^{\theta'}$.

Подстановка θ **замкнута**, если $\forall x \in Var(\theta) \quad Var(x^\theta) = \emptyset$. Множество всех замкнутых подстановок из $\Theta(X)$ обозначается X^\bullet .

Пусть заданы терм $e \in Tm$ и список $\vec{x} = (x_1, \dots, x_n)$ различных переменных, причём $Var(e) \subseteq \{x_1, \dots, x_n\}$. Будем использовать обозначения:

- $D_{\tau(\vec{x})}$ обозначает множество $D_{\tau(x_1)} \times \dots \times D_{\tau(x_n)}$,
- $e(\vec{x})$ обозначает функцию вида $D_{\tau(\vec{x})} \rightarrow D_{\tau(e)}$, такую, что

$$\forall \vec{d} = (d_1, \dots, d_n) \in D_{\tau(\vec{x})} \quad e(\vec{x}) : \vec{d} \mapsto e^{(d_1/x_1, \dots, d_n/x_n)}. \quad (1.4)$$