

Оглавление

1	Введение	13
1.1	Проблема верификации программ	13
1.2	Необходимые математические понятия	16
1.2.1	Термы и связанные с ними понятия	16
1.2.2	Примеры типов и функциональных символов	18
1.2.3	Подстановки	20
1.2.4	Массивы	21
1.2.5	Истинностные значения утверждений	22
I	Верификация последовательных и распределённых программ	23
2	Программы, представленные в виде блок-схем	25
2.1	Понятие блок-схемы	25
2.2	Выполнение блок-схемы	27
2.3	Примеры блок-схем	28
2.4	Задача верификации блок-схем	31
3	Метод инвариантов для верификации блок-схем	33
3.1	Базовые множества и базовые пути	33
3.2	Описание метода инвариантов	34
3.3	Обоснование метода инвариантов	35
3.4	Примеры фундированных множеств	36
3.5	Применение метода инвариантов	37
3.5.1	Верификация блок-схемы вычисления суммы	37
3.5.2	Верификация блок-схемы деления с остатком	39
3.5.3	Верификация блок-схемы извлечения корня	40
3.5.4	Верификация блок-схемы возведения в степень	42
3.5.5	Верификация блок-схемы сортировки	42

4	Процесные представления блок-схем	47
4.1	Понятие процесса	47
4.1.1	Действия	47
4.1.2	Процессы и их выполнение	48
4.2	Процессы, соответствующие блок-схемам	49
4.3	Верификация процесных представлений блок-схем	50
5	Верификация операторных программ	53
5.1	Понятие операторной программы	53
5.2	Примеры операторных программ	54
5.3	Метод инвариантов для верификации операторных программ	55
5.4	Пример верификации операторной программы	56
6	Распределенные программы	59
6.1	Понятие распределенной программы	59
6.1.1	Действия	59
6.1.2	Процессы	60
6.1.3	Распределенные программы	61
6.1.4	Каналы в распределенных программах	61
6.1.5	Переходы в распределенных программах	62
6.1.6	Выполнение распределенной программы	64
6.2	Спецификация и верификация распределенных программ	65
6.3	Примеры распределенных программ	66
6.3.1	Вычисление факториала	66
6.3.2	Передача сообщений через буфер	67
6.3.3	Избрание лидера	68
6.3.4	Параллельная сортировка	69
6.4	Распределенная программа перемножения матриц	70
6.4.1	Неформальное описание распределенной программы перемножения матриц	70
6.4.2	Спецификация программы перемножения матриц	70
6.4.3	Определение распределённой программы перемножения матриц	71
6.4.4	Дополненные процессы	72
6.4.5	Верификация программы перемножения матриц	73
7	Задачи и исследовательские проблемы	79
7.1	Верификация программ без массивов	79
7.1.1	Произведение двух чисел	79
7.1.2	Возведение в степень	80

7.1.3	Извлечение квадратного корня	80
7.1.4	Извлечение логарифма	81
7.1.5	Вычисление частного и остатка от деления целых чисел	81
7.1.6	Наибольший общий делитель	82
7.1.7	Представление наибольшего общего делителя линейной формой	84
7.1.8	Наибольший общий делитель и наименьшее общее кратное	84
7.1.9	Приближенное решение уравнения	85
7.1.10	Проверка на простоту	85
7.1.11	Проверка, является ли число совершенным	86
7.2	Верификация программ с массивами	86
7.2.1	Инвертирование массива	86
7.2.2	Минимальный элемент массива	87
7.2.3	Двоичный поиск	87
7.2.4	Наибольший общий делитель компонентов массива	88
7.2.5	Список простых чисел от 2 до n	88
7.2.6	Сортировка массива	89
7.2.7	Перестановка массива с заданным условием	90
7.2.8	Перестановка массива в заданном порядке	90
7.2.9	Вычисление определителя матрицы	91
7.3	Задачи на составление программ	92
7.4	Верификация распределенных программ	94
7.5	Исследовательские проблемы	95

II Верификация функциональных программ 97

8	Введение в функциональное программирование	99
8.1	Парадигма функционального программирования	99
8.2	Примеры функциональных программ	100
8.2.1	Конкатенация строк	101
8.2.2	Инвертирование строки	101
8.2.3	Поиск подстроки	102
9	Функциональные программы	103
9.1	Пополненные домены и функции на них	103
9.1.1	Пополненные домены	103
9.1.2	Монотонные функции	103
9.1.3	Естественные продолжения	104

9.1.4	Частично упорядоченные множества монотонных функций	105
9.1.5	Полные частично упорядоченные множества	106
9.2	Функциональные программы	106
9.2.1	Понятие функциональной программы	106
9.2.2	Функционал, соответствующий функциональной программе	107
9.2.3	Непрерывные функционалы на полных частично упорядоченных множествах	107
9.2.4	Неподвижные точки функционалов на полных частично упорядоченных множествах	108
9.2.5	Непрерывность функционалов, соответствующих функциональным программам	109
9.2.6	Нахождение наименьших неподвижных точек функциональных программ	112
9.2.7	Примеры неподвижных точек функциональных программ	113
9.2.8	Немонотонные функциональные программы	114
9.3	Алгоритмическая полнота функциональных программ	115
10	Вычисление значений наименьших неподвижных точек	117
10.1	Постановка задачи	117
10.2	Метод решения	117
10.3	Вычислительные правила	119
10.4	Упрощение терма	119
10.5	Функция C_{Σ}	124
10.6	Вспомогательные понятия	126
10.6.1	Полные раскрытия	126
10.6.2	Индексированные термы	128
10.6.3	Σ -переходы	128
10.7	Безопасные вычислительные правила	131
10.8	Свойства правил PO, LO, PI, LI	137
10.8.1	Безопасность правила PO	137
10.8.2	Безопасность правила LO	139
10.8.3	Пример небезопасности правила LO	141
10.8.4	Пример небезопасности правил PI и LI	141
11	Верификация функциональных программ	143
11.1	Задача верификации функциональных программ	143
11.2	Метод вычислительной индукции	143
11.2.1	Описание метода	143

11.2.2	Примеры верификации функциональных программ методом вычислительной индукции	145
11.3	Метод структурной индукции	148
11.3.1	Описание метода	148
11.3.2	Примеры верификации функциональных программ методом структурной индукции	149
11.4	Другие методы верификации функциональных программ	155
11.4.1	Оценка наименьшей неподвижной точки функциональной программы сверху	155
11.4.2	Эквивалентные преобразования функциональных программ	156
12	Задачи и исследовательские проблемы	161
12.1	Нахождение наименьших неподвижных точек функциональных программ	161
12.2	Вид наименьших неподвижных точек	161
12.3	Функции, определяемые функциональными программами	163
12.4	Свойства наименьших неподвижных точек	166
12.5	Исследовательские проблемы	169
III	Model checking	171
13	Модели распределённых программ	173
13.1	Верификация распределённых программ	173
13.1.1	Математические модели распределённых программ	173
13.1.2	Спецификация	174
13.1.3	Построение формальных доказательств	175
13.2	Системы переходов	175
13.2.1	Понятие системы переходов	176
13.2.2	Пути в системах переходов	176
13.2.3	Построение системы переходов, соответствующей распределенной программе	177
14	Model checking на основе CTL	181
14.1	Темпоральная логика CTL	181
14.1.1	Формулы темпоральной логики CTL	181
14.1.2	Значения CTL-формул	182
14.1.3	Эквивалентность CTL-формул	182
14.1.4	Примеры свойств распределённых программ, выражаемых CTL-формулами	184

14.2	Задача model checking для CTL	185
14.3	MC-CTL на основе понятия неподвижной точки	186
14.3.1	Неподвижные точки монотонных операторов	186
14.3.2	Вычисление множеств $(\mathbf{EU}(B, C))^S$ и $(\mathbf{EGB})^S$ на основе понятия неподвижной точки	187
14.3.3	Алгоритм решения задачи MC-CTL на основе поня- тия неподвижной точки	190
14.4	μ -исчисление	190
14.4.1	μ -формулы	190
14.4.2	Значения μ -формул	191
14.4.3	Ускоренное вычисление значений μ -формул	195
14.4.4	Вложение CTL в μ -исчисление	196
15	Бинарные диаграммы решений	199
15.1	Бинарные диаграммы решений и их свойства	199
15.1.1	Определение бинарной диаграммы решений	199
15.1.2	Эквивалентность и изоморфность бинарных диаграмм решений	200
15.1.3	Представление множеств замкнутых подстановок би- нарными диаграммами решений	200
15.1.4	Редукция бинарных диаграмм решений	201
15.1.5	Представление булевых функций	202
15.1.6	Подстановка значений вместо переменных	202
15.2	Согласованность с порядком переменных	203
15.3	Алгебраические операции на BDD	206
15.3.1	Булевы операции	206
15.3.2	Произведение	207
15.4	MC-CTL с использованием бинарных диаграмм решений	209
15.5	Оптимизирующие преобразования	211
16	Model checking на основе LTL	213
16.1	Формулы темпоральной логики LTL	213
16.2	Квантифицированные LTL-формулы	214
16.3	Задачи model checking для LTL	215
16.3.1	Система переходов Σ_A	215
16.3.2	Система переходов $\Sigma \times \Sigma_A$	217
16.3.3	Первая задача model checking для LTL	219
16.3.4	Вторая задача model checking для LTL	220
16.4	Автоматы Бюхи	223
16.4.1	Понятие автомата Бюхи	223
16.4.2	Язык автомата	223

16.4.3	Эквивалентность автоматов	224
16.4.4	Пересечение автоматов	225
16.4.5	Использование автоматов Бюхи для MC-LTL	226
16.4.6	Оптимизация построения автомата $\mathcal{B}_{\bar{A}}$	226
17	Вероятностный model checking	229
17.1	Введение	229
17.2	Вероятностные системы переходов	230
17.2.1	Понятие вероятностной системы переходов	230
17.2.2	Примеры вероятностных систем переходов	231
17.3	Темпоральная логика PCTL	233
17.3.1	Свойства вероятностных систем переходов	233
17.3.2	Формулы логики PCTL	234
17.3.3	Значения формул логики PCTL в состояниях вероятностных систем переходов	234
17.3.4	Интерпретация значений формул логики PCTL	236
18	Задачи и исследовательские проблемы	237
18.1	Задачи	237
18.2	Исследовательские проблемы	238
IV	Теория процессов	241
19	Понятие процесса	243
19.1	Неформальное понятие процесса и примеры процессов	243
19.1.1	Неформальное понятие процесса	243
19.1.2	Пример процесса	244
19.1.3	Другой пример процесса	245
19.2	Действия	246
19.3	Определение понятия процесса	247
20	Операции на процессах	251
20.1	Префиксное действие	251
20.2	Пустой процесс	252
20.3	Альтернативная композиция	252
20.4	Параллельная композиция	255
20.5	Ограничение	257
20.6	Переименование	259
20.7	Свойства операций на процессах	259

21 Эквивалентности процессов	263
21.1 Простая эквивалентность	263
21.1.1 Поведение процесса	263
21.1.2 Понятие простой эквивалентности	264
21.1.3 Примеры процессов, находящихся в отношении простой эквивалентности	264
21.2 Сильная эквивалентность	265
21.2.1 Понятие сильной эквивалентности	265
21.2.2 Критерий сильной эквивалентности, основанный на понятии бимоделирования	266
21.2.3 Логический критерий сильной эквивалентности	268
21.2.4 Алгебраические свойства сильной эквивалентности	270
21.2.5 Распознавание сильной эквивалентности	271
21.2.6 Минимизация процессов относительно сильной эквивалентности	273
21.3 Наблюдаемая эквивалентность	277
21.3.1 Определение наблюдаемой эквивалентности	277
21.3.2 Критерий наблюдаемой эквивалентности, основанный на понятии наблюдаемого бимоделирования	278
21.3.3 Логический критерий наблюдаемой эквивалентности процессов	279
21.3.4 Алгебраические свойства наблюдаемой эквивалентности	279
21.3.5 Распознавание наблюдаемой эквивалентности	280
21.4 Наблюдаемая конгруэнция	281
21.4.1 Мотивировка наблюдаемой конгруэнции	281
21.4.2 Определение наблюдаемой конгруэнции	282
21.4.3 Связь между наблюдаемой эквивалентностью и наблюдаемой конгруэнцией	283
21.5 Другие эквивалентности процессов	285
22 Примеры доказательства свойств процессов	287
22.1 Мастерская	287
22.2 Планировщик	289
23 Процессы с передачей сообщений	293
23.1 Действия с передачей сообщений	293
23.2 Процессы с передачей сообщений	293
23.2.1 Операторы	293
23.2.2 Понятие процесса с передачей сообщений	294
23.2.3 Операции на процессах	295

23.2.4	Редукция процессов	296
23.3	Наблюдаемая эквивалентность процессов	296
23.3.1	Система переходов процесса	296
23.3.2	Метод доказательства наблюдаемой эквивалентности процессов	297
23.4	Примеры верификации процессов с передачей сообщений	301
23.4.1	Последовательная композиция буферов	302
23.4.2	Разделение мультимножеств	305
23.4.3	Вычисление квадрата	309
24	Процессы с асинхронным взаимодействием	313
24.1	Понятие асинхронного взаимодействия	313
24.2	Понятие процесса с асинхронным взаимодействием	314
24.3	Протоколы передачи данных	315
24.3.1	Однонаправленный протокол с чередующимися битами	315
24.3.2	Двунаправленный протокол передачи сообщений с чередующимися битами	318
24.3.3	Протокол скользящего окна с возвратом	320
24.3.4	Протокол скользящего окна с заданным повтором	322
25	Задачи и исследовательские проблемы	329

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 обоих 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. Например, две программы P_1 и P_2 , работающие с совместно используемым ресурсом R , могут использовать в качестве семафора общую переменную b , принимающую 2 значения: 1 (доступ к R открыт) и 0 (доступ к R закрыт). Если какая-либо из этих программ, например P_1 , хочет работать с R , она проверяет значение b :

- если $b = 1$, то P_1 изменяет значение b на 0 (запрещая тем самым доступ к R программе P_2 на время своей работы с R) и работает с R , после чего изменяет значение b обратно на 1,
- а если $b = 0$, то P_1 ждет, пока значение b не станет равным 1.

Однако если программа P_2 тоже хочет работать с R , и она проверила значение b после того момента когда его проверила P_1 , но до того момента как P_1 изменила значение b с 1 на 0, то может возникнуть нарушение дисциплины доступа к R .

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 .

Символ \mathcal{D} обозначает множество всех значений всех типов.

- 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

если e_1 и e_2 – различные подтермы терма e , то либо $e_1 \subset e_2$,
либо $e_2 \subset e_1$, либо e_1 и e_2 не имеют общих компонентов. (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 $E \cap Tm_\tau$,
- 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** данного 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 \mathbf{B} , и $D_{\mathbf{B}} = \{0, 1\}$. Термы типа \mathbf{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 булевы 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 $\left\{ \begin{array}{c} e_1 \\ \vdots \\ e_n \end{array} \right\}$ и $\left[\begin{array}{c} e_1 \\ \vdots \\ e_n \end{array} \right]$ соответственно, а также в виде $\bigwedge_{i \in \{1, \dots, n\}} e_i$ и $\bigvee_{i \in \{1, \dots, n\}} e_i$ соответственно, и также в виде $\{e_1, \dots, e_n\}$ и $[e_1, \dots, e_n]$ 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 $if\ e\ then\ e_1\ else\ e_2$, или $e?e_1 : e_2$.

Строковые типы и функциональные символы

Types содержит типы **L** 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 $\mathbf{S} \rightarrow \mathbf{L}$ и $\mathbf{S} \rightarrow \mathbf{S}$ 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 $(\mathbf{L}, \mathbf{S}) \rightarrow \mathbf{S}$, 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 $(\tau_1, \dots, \tau_n) \rightarrow (\tau_1, \dots, \tau_n)$, ему 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

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

1.2.4 Массивы

В программах могут использоваться структуры данных, называемые **массивами**. Массивы обозначаются записями вида $a_{m..n}$, где a – имя массива, m и n – термы типа \mathbf{I} , обозначающие нижнюю и верхнюю границы массива соответственно. Массив $a_{m..n}$ может обозначаться более коротко путем указания лишь его имени, без указания нижней и верхней границ. Если значения m, n нижней и верхней границ массива a таковы, что $m \leq n$, то массив a непуст, в противном случае он является пустым. Для каждого массива $a_{m..n}$ и каждого $i \in \{m, \dots, n\}$ определен объект, обозначаемый записью a_i и называемый **компонентой** массива a с индексом i . Компоненты массива a рассматриваются как переменные одинакового типа. Если a и b – массивы вида $a_{m..n}$ и $b_{m..n}$, то $b = perm(a)$ означает, что b – перестановка a , т.е. существует биекция f на $\{m, \dots, n\}$, такая, что $\forall i \in \{m, \dots, n\} \quad b_i = a_{f(i)}$.

Пусть задан массив $a_{m..n}$ и $i, j, i', j' \in \{m, \dots, n\}$. Будем обозначать записями $ord(a_{i..j})$, $a_{i..j} \leq a_{i'..j'}$, $a_{i..j} \leq a_{i'}$ формулы соответственно:

$$\bigwedge_{i \leq k < j} (a_k \leq a_{k+1}), \quad \bigwedge_{\substack{i \leq k \leq j \\ i' \leq k' \leq j'}} (a_k \leq a_{k'}), \quad \bigwedge_{i \leq k \leq j} (a_k \leq a_{i'}).$$

(Напомним, что если множество конъюнктивных членов пусто, то их конъюнкция равна 1.)

1.2.5 Истинностные значения утверждений

Будем использовать следующее обозначение: если A – произвольное утверждение (выраженное на естественном или формальном языке), то запись $\llbracket A \rrbracket$ обозначает значение 1 если A истинно и 0 если A ложно.

Конец ознакомительного фрагмента.

Приобрести книгу можно

в интернет-магазине

«Электронный универс»

e-Univers.ru