Оглавление

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
  $$(\tau_1, \ldots, \tau_n) \to \tau, \tag{1.1}$$
  где $\tau_1, \ldots, \tau_n, \tau \in Types$, и

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

  (Напомним, что функция $f : A \to B$ 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 $e_1, \ldots, e_n \in Tm$, $f \in Fun \cup FVar$, и $\tau(f)$ имеет вид (1.1), где $\tau_1 = \tau(e_1)$, …, $\tau_n = \tau(e_n)$, то запись $f(e_1, \ldots, e_n)$ – терм типа τ.

Терм $e \in Tm$ называется **подтермом** терма $e' \in Tm$, если либо $e = e'$, либо $e' = f(e_1, \ldots, e_n)$, где $f \in Fun \cup FVar$, и $\exists\, i \in \{1, \ldots, n\}$: 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

$$\text{если } e_1 \text{ и } e_2 - \text{различные подтермы терма } e, \text{ то либо } e_1 \subset e_2, \text{ либо } e_2 \subset e_1, \text{ либо } e_1 \text{ и } e_2 \text{ не имеют общих компонентов.} \tag{1.2}$$

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 $\{x \in Var \mid x \in e\}$,
- $\forall\, e_1, \ldots, e_n \in Tm\ Var(e_1, \ldots, e_n) = Var(e_1) \cup \ldots \cup Var(e_n)$,
- $\forall\, X \subseteq Var\ Tm(X)$ 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 $(\tau) \to \tau'$,
- для каждой рассматриваемой функции $f : E \to E'$, 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 $f(e_1, \ldots, e_n)$, то $eval(e)$ определён, только если определено значение функции f на кортеже $(eval(e_1), \ldots, eval(e_n))$, 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 $\mathbf{N}$ и $\mathbf{I}$, значениями которых являются натуральные $(0, 1, \ldots)$ и целые числа соответственно.

Fun содержит ФС $+, -, \cdot, div, mod$ типа $(\mathbf{I}, \mathbf{I}) \to \mathbf{I}$, 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 $e_1 \wedge \ldots \wedge e_n$ и $e_1 \vee \ldots \vee e_n$ могут также записываться в виде $\left\{ \begin{smallmatrix} e_1 \\ \vdots \\ e_n \end{smallmatrix} \right\}$ и $\left[\begin{smallmatrix} e_1 \\ \vdots \\ e_n \end{smallmatrix} \right]$ соответственно, а также в виде $\bigwedge_{i \in \{1,\ldots,n\}} e_i$ и $\bigvee_{i \in \{1,\ldots,n\}} e_i$ соответственно, и также в виде $\{e_1, \ldots, e_n\}$ и $[e_1, \ldots, e_n]$ 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 $\mathbf{L}$ и $\mathbf{S}$, 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 $\mathbf{S} \to \mathbf{L}$ и $\mathbf{S} \to \mathbf{S}$ 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 $(\mathbf{L}, \mathbf{S}) \to \mathbf{S}$, 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 $\tau_1, \dots, \tau_n$ (некоторые компоненты этого списка могут совпадать)

- $Types$ содержит тип, обозначаемый записью $(\tau_1, \dots, \tau_n)$, и

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

- Fun содержит ФС $tuple$ типа $(\tau_1, \dots, \tau_n) \to (\tau_1, \dots, \tau_n)$, 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 $\theta : Var \to Tm$. 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

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

  вторая запись в (1.3) используется, когда $Var(\theta) = \{x_1, \ldots, x_n\}$;
- $\forall\,\theta \in \Theta$, $\forall\, e \in Tm$ 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 $\forall\, x \in Var(\theta)\ Var(x^\theta) = \emptyset$. Множество всех замкнутых подстановок из $\Theta(X)$ обозначается $X^\bullet$.

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

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

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

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 **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 $i \in \{m, \ldots, n\}$ 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 $\{m, \ldots, n\}$, такая, что $\forall\, i \in \{m, \ldots, n\}\ \ b_i = a_{f(i)}$.

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

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

(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 $[\![A]\!]$ обозначает значение 1 если A истинно и 0 если A ложно.