Машины Абстрактных Состояний
(Машины Гуревича)

И.П. Соловьев

Основные понятия

В сравнении с методами, ранее рассмотренными в курсе ТВПС, это «значительно более или вполне операционный» метод спецификации алгоритмов и структур данных (программных проектов) — в терминах абстр вычисл у-ва.

(История: Гуревич – Екатеринбург – Мичиган - Майкрософт)

В данной главе изучается понятие Машины Абстрактных Состояний, в более ранней терминологии — е-алгебры или «развивающейся» алгебры (EA). В англоязычной литературе используются, соответственно, термины Abstract State Machine (ASM) и Evolving Algebras.

Это понятие появилось и было развито в работах Ю.Гуревича и его коллег и учеников относительно недавно — в начале 90-х годов. ASM иногда называют просто машинами Гуревича. Далее в тексте в случаях, когда это не вызывает неоднозначного толкования, вместо термина ASM иногда используется термин (абстрактная) машина.

основные цели данного подхода

- предложить формально строгий метод спецификации,
- с ясной нотацией,
- с подходящей и удобной инструментальной поддержкой проектирования и

анализа систем высокого уровня,

- предоставить средства построения т.наз. исполняемых спецификаций или исполняемых моделей программных систем, т.е. спецификации ASM могут быть выполнены при помощи подходящего интерпретатора,
- спецификации на различных уровнях абстракции, т.е. в качестве средства многоуровневого проектирования, др.словами, это разновидность применение метода последовательных уточнений,
- в частности, ASM должны использоваться для моделирования межкомпонентного взаимодействия на любом желаемом уровне детализации прогр проекта,
- построенные модели абстрактных машин должны быть использованы в течение всего жизненного цикла прогр обесп,
- проектирование спецификаций н терминах предметной области применительно к требуемому уровню абстракции (т.е. в терминах функций и отношений данной предметной области),
- и как средства доказательства или проверки корректности реализации вычислительных систем,
- средство разработки быстрых прототипов — постепенным уточнением спецификация преобразуется в целевой код (возможно, с использ доп технологич расширений).

Понятие ASM легло в основу языка выполняемых спецификаций высокого уровня (ASML и др.), предназначенного для использования в качестве метода определения операционной семантики вычислительных систем, в частности, языков программирования. Важно то, что синтаксически этот язык близок к широко распространенным императивным языкам программирования, таким как Паскаль и Си, и может быть легко освоен любым практикующим программистом. Как следствие, это может повысить качество программного продукта и понизить сроки и стоимость проектирования.

Основная цель описываемого подхода состоит в создании операционной семантики или механизма описания вычислений алгоритмов посредством развития того, что можно было бы назвать тезисом Тьюринга: каждый алгоритм может быть вычислен подходящей машиной Тьюринга [1] (точнее, Тьюринг говорил о вычислимых функциях, но в данном контексте это несущественно). Здесь также уместно вспомнить и о тезисе Черча.

Непосредственное использование машин Тьюринга в качестве метода определения операционной семантики алгоритмов представляется не слишком удобным и даже неуклюжим. Один шаг некоторого алгоритма может потребовать длинной последовательности шагов моделирующей машины Тьюринга. Более целесообразно моделировать поведение алгоритмов «шаг в шаг» (неформальное требование), т.е. так, чтобы один шаг алгоритма требовал ограниченное число шагов

моделирующей машины.

Существуют и другие методы моделирования поведения алгоритмов, свободные от описанного недостатка машин Тьюринга, например машины с произвольным доступом к памяти. Однако все они имеют фиксированный уровень абстракции, который может оказаться слишком низким для некоторых алгоритмов. Хорошим примером может быть реализация абстрактного типа данных «стек».

В свою очередь метод ASM позволяет создавать целую иерархию моделирующих машин на различных уровнях абстракции того или иного алгоритма. Идея такого моделирования очень проста. Сложные шаги, требующие большого количества (простых вычислительных) операций, моделируются одним оператором. Например, на нескольких уровнях абстракции построены иерархии ASM для определения семантики таких языков как Си [10], Си++ [15] и Ява (Java) [16].

Интересно отметить, что автор применения идеи ASM (EA) для описания операционной семантики Ю.Гуревич предлагает также рассматривать этот аппарат как еще одну формализацию понятия алгоритма. Уже накопленный опыт и интуиция показывают (но, разумеется, не доказывают), что каждый алгоритм может быть смоделирован подходящей ASM. При этом моделирующая машина находится на том же уровне абстракции, что и данный алгоритм, и не использует ресурсов «значительно больше» чем данный алгоритм. Моделирование выполняется «шаг в шаг».

Подтверждением данного тезиса, в частности, может служить тот факт, что уже

построены ASM для языков Си, Си++, Ява, Модула-2 [11], Пролог [5] и др.

Приведенные выше рассуждения, в основном, относились к последовательным алгоритмам и, соответственно, к последовательным ASM. Однако это понятие обобщается и на распределенные или параллельные (непоследовательные) алгоритмы. Примерами могут служить абстрактные машины, построенные для языков Оккам [12], Парлог [4] и Параллельный Пролог [3].

Машины Гуревича относительно легки для понимания и конструирования. Их можно использовать в учебной аудитории. Даже небольшие программы, написанные на необычных языках, могут быть трудны для непосредственного восприятия. В такой ситуации может оказать помощь даже набросок ASM, сделанный на доске.

В Мичиганском университете под руководством Ю.Гуревича был написан на языке Си интерпретатор машин абстрактных состояний [10]. По заданной абстрактной машине этот интерпретатор может выполнить указанное число шагов и затем проверить полученное состояние, либо остановить работу машины в момент приобретения некоторым предикатом значения «истина», и т.д. Известны и другие примеры. В частности, под руководством автора в С.-Петербургском университете аспирант Усов А.А. разработал интернет-ориентированный интерпретатор ASM [www.math.spbu.ru/~soloviev], который способен выполнять и отлаживать спецификации алгоритмов как на локальном компьютере, так и на удаленном сервере.

Применение аппарата ASM не ограничивается учебной аудиторией. Помимо

определения операционной семантики известных языков программирования эти машины используются в теории баз данных и ее приложениях, например для спецификации возможностей объектно-ориентированных моделей баз данных [13]. Известны примеры применения машин Гуревича для обоснования корректности реализации вычислительных систем [14], для спецификации микропроцессорных схем [2] и т.п.

Рассмотрим основную идею, заложенную в понятие абстрактной машины Гуревича. Предположим сначала, что рассматривается случай последовательных алгоритмов. Это означает, что время изменяется последовательно, алгоритм начинает свою работу в некотором начальном состоянии $\mathbf{S_0}$ и продолжает свою работу, проходя последовательно состояния $\mathbf{S_1}$, $\mathbf{S_2}$ и т.д., причем на каждом шаге выполняется ограниченное количество работы. Каждое состояние алгоритма может быть представлено (алгебраической) структурой — некоторым множеством с отношениями и функциями (отсюда и возникло первоначальное название — развивающаяся алгебра). (Здесь описывается модель с дискретным временем, но существуют модификации и с непрерывным временем.)

Таким образом, выполнение алгоритма представляется в виде последовательности описанных структур. Однако одного такого представления недостаточно. Необходимо также указать, как происходит переход из состояния $\mathbf{S_i}$ в состояние $\mathbf{S_{i+1}}$. Для того чтобы совершить такой переход, выполняется некоторое

ограниченное количество изменений в состоянии $\mathbf{S_i}$. В результате оказывается, что алгоритм в целом может быть представлен в виде конечного числа правил перехода очень простого вида.

На отдельные состояния алгоритма можно смотреть как на нечто достижимое и выполнимое; их можно представить на любом компьютере, корректно реализующем этот алгоритм, в то время как весь процесс в целом может оказаться слишком громоздким для такого представления.

В последние годы область применения ASM активно расширяется (особенно для спецификации распределенных систем и аппаратных схем). Развивается и само понятие ASM, так что его нельзя считать окончательно сформировавшимся. Для более полного представления о текущем состоянии этой области рекомендуем обратиться к приведенному списку литературы или к персональной странице Ю.Гуревича на веб-сайте мичиганского университета: http://www.eecs.umich.edu/~gurevich или http://www.eecs.umich.edu/~gasm.

Синтаксис

Введем в рассмотрение математические объекты, необходимые для описания любых состояний произвольных алгоритмов на наиболее естественном для них уровне абстракции [9]. В качестве наиболее подходящего для этой цели средства математикая логика предлагает ***алгебры*** (или ***структуры***) — множества с функциями и операциями или функциями и отношениями (универсальная алгебра — это структура 1го порядка без отношений — отношения моделируются функциями). Незначительная модификация позволяет применять это понятие как в статических, так и в динамических ситуациях. Модифицированные подходящим образом структуры называются ***состояниями***.

Прежде всего, зафиксируем используемые базовые синтаксические конструкции (сигнатуру) — конечный набор функциональных имен (символов) — функторов, и счетный набор переменных.

Каждому функциональному имени сопоставляется некоторое число аргументов или размерность. Пара (функциональное имя f, число аргументов n), называется функтором и обозначается f/n.

Конечное множество функторов называется ***словарем***.

Для обозначения словарей обычно используется греческая буква ϒ.

Некоторые функторы особым образом помечаются как имена ***отношений*** (***предикатные*** символы) и как ***статические*** (или динамические**) (**см. ниже**)**.

Имена функций с областью значений {**true, false**} трактуются как имена отношений (предикатов).

Предполагается, что каждый словарь содержит следующие логические имена функций (и отношений): символ равенства =/2, нульместные функциональные имена **true, false,** предикат Bool/1 и имена обычных булевских операций. Все логические имена являются именами отношений (булевозначных функций). Словарь также включает специальный символ **undef** или **ω** Все указанные предопределенные имена, включая символ **undef**, по определению считаются статическими.

Примеры

1. Словарь ориентированных деревьев: предикат *Nodes*/1, функторы *Parent*/1, *FirstChild*/1, *NextSibling*/1 и *C*/0 (подразумевается *Current Node*).
2. Словарь размеченных (строковых) ориентированных деревьев: к первому словарю добавляются предикатный функтор *String*/1 и функтор *Label*/1.
3. *Push*/2

Термы определяются рекурсивно, как в логике первого порядка:

– переменная есть терм;

– если f/n — функтор и $t_1, \ldots, t_n$ — термы, то $f(t_1,\ldots,t_n)$ — терм.

При $n = 0$ допускаются записи $f()$ или просто f.

Термы, не содержащие переменных, называются ***основными***.

Пусть далее запись вида $\overline{t}$ обозначает кортеж (конечная последовательность) некоторых термов $t_1,\ldots,t_n$ *длины* или *размерности n*, причем размерность каждый раз

определяется контекстом, в частности, запись вида $f(\overline{t})$ всегда означает, что размерности функтора и кортежа совпадают.

Если старший функтор f терма $f(\overline{u})$ является предикатным символом, то терм $f(\overline{t})$ называется булевским.

П р и м е р ы

Несколько примеров термов в словаре ориентированных деревьев:

NextSibling(FirstChild)

Parent(*C*) = **undef**

Parent(*FirstChild*(*C*)) = *C*

Parent(*C*) = *Parent*(*NextSibling*(*C*))

Push(Head(F),S)

Состояния

Определим понятие ***состояния*** абстрактной машины — подразумевается состояние вычислений (первоначально для состояния использовался термин ***статическая алгебра***).

Пусть даны непустое множество X, словарь Υ и интерпретация функциональных имен из Υ — некоторое отображение $\boldsymbol{I} : \Upsilon \to \{X^n \to X\}$, такое, что каждому n-местному функтору ставится в соответствие некоторая всюду определенная n-местная функция. Тогда пара (X,I) называется ***состоянием*** S некоторой ASM над словарем Υ (с интерпретацией I).

Пусть далее $\mathrm{Voc}(S) = \Upsilon$.

Множество X называется ***базисным множеством*** или ***суперуниверсумом*** состояния S и обозначается $\mathrm{BaseSet}(S)$. Подмножества X называются ***универсумами***, элементы X также называются *элементами* S.

Функции, интерпретирующие функторы из Υ, называются ***базисными функциями*** S. Если $\boldsymbol{f}$ — функтор, то $\boldsymbol{f_S}$ (иногда $\boldsymbol{f_I}$) обозначает его интерпретацию (базисную функцию) в состоянии S.

В частном случае, нульместные имена интерпретируются элементами из X. Базисные нульместные функции иногда называются также ***выделенными***

элементами, поскольку их значения выделяют (указывают на) некоторые элементы суперуниверсума (в некот смысле б вып роль перем). Выделенный элемент *f*, принимающий свое значение в универсуме *U*, обозначается *f*: *U*.

Несколько замечаний к введенным понятиям:

- интерпретация предопр символов **true, false, undef** различаются; в то же время не делается различий между каждым из этих символов и его интерпретацией;
- интерпретациями предикатных символов типа *P/n* являются функции типа $X^n \to$ {**true, false**} **—** базисные отношения состояния *S*;
- на базисные отношения следует смотреть как на множества кортежей $\bar{x}$, таких, что $P(\bar{x})$ = **true**;
- одноместное отношение (как и соответствующее подмножество) называется ***универсумом***; представление отношений булевозначными функциями вызвано соображениями удобства выполнения модификаций состояний вычислений (см. ниже);
- пусть **Bool** обозначает универсум (отношение) {**true, false**};
- булевские операции интерпретируются обычным образом;
- знак равенства интерпретируется как отношение тождества на *X*.

П р и м е р ы

Приведем примеры состояния машин (статических алгебр).

1. Ориентированное дерево с n узлами порождает состояние с $n + 3$ элементами. В дополнение к n узлам базисное множество содержит обязательные логические элементы **true**, **false**, **undef**. Универсум *Nodes* (определяемый с помощью предиката *Nodes*/1) содержит n узлов и не содержит логические элементы.

2. Предположим, что словарь содержит нульместные функции (константы) 0, 1 и бинарные функторы + и ×. Суперуниверсум образует множество целых чисел. Функциональные имена интерпретируются обычным для арифметики образом.

Замечание к роли элемента **undef** или **ω**. С формальной точки зрения определенные выше базисные функции являются всюду определенными. Такое ограничение может показаться слишком сильным. Во многих случаях представляется естественным использование частичных функций. Например, можно было бы попытаться расширить определенную выше арифметическую структуру операцией деления, частичной на множестве целых чисел.

Вместо обобщения понятия состояния за счет введения частичных функций применим специальный прием, опирающийся на использования выделенного элемента **undef**. Положим в качестве области определения базисной n-местной функции f множество Dom(f) всех кортежей $\bar{x}$ размерности n, таких, что $f(\bar{x}) \neq$ **undef** (подчеркнем еще раз, что **undef —** обычный элемент суперуниверсума). Тогда, с некоторой долей условности, базисные функции можно считать частичными.

Вернемся к примеру арифметической машины и введем такую операцию деления ÷, что значение $a \div b \neq \omega$ для целых a и b, таких, что b делит a, и только для них.

Еще одно замечание относится к реализации механизма обработки данных различных типов. Один из самых распространенных способов решения этой проблемы — обобщение понятия состояния машины (статической алгебры) до так называемой ***многосортной алгебры***. Однако в этом нет необходимости. Достаточно воспользоваться понятием универсума — множества элементов суперуниверсума, удовлетворяющих некоторому заданному одноместному отношению (свойству).

С учетом последнего замечания уточним роль универсума Bool: результат любой булевской операции есть ω, если хотя бы один из аргументов не принадлежит этому универсуму (числе и ‘=’/2, т.е. используется слабое равенство).

Таким же образом модифицируется пример с арифметической машиной: достаточно ввести универсум *Integer*, состоящий из множества всех целых чисел. Пользуясь тем же приемом, можно ввести, например, отношение порядка < на множестве целых. Аналогично любое отношение R, определенное на некотором универсуме U, может быть представлено ***характеристической функцией*** R — функцией, неопределенной (т.е. равной **undef**), если по крайней мере один из ее аргументов не принадлежит U.

Абстракция взятия памяти. Многие алгоритмы во время исполнения требуют дополнительной памяти. В более абстрактном смысле это означает, что состояния требуют построения новых элементов. Возможно (и удобно) иметь источник новых элементов внутри состояния. Для этого необходимо предположить наличие дополнительного бесконечного (или достаточно большого конечного) универсума ***Reserve*** зарезервированных элементов, непересекающегося с другими универсумами, и некоторого способа выбора этих элементов.

Определим ***резерв*** состояния S как множество, содержащее все элементы S, такие, что

- каждое базисное отношение, за исключением равенства, при вычислении порождает **false**, если хотя бы один аргумент принадлежит резерву;
- каждая базисная функция (не отношение) при вычислении порождает **undef**, если хотя бы один аргумент принадлежит резерву;
- множества значений базисных функций не содержат элементов резерва.

Далее мы будем предполагать, что каждое состояние имеет бесконечный резерв.

Значения переменных

Рассмотрим назначение значений переменным в некотором состоянии S — *означивание*. Означивание переменных в S есть функция ζ из некоторого множества переменных в множество BaseSet(S). Например, для булевской переменной v значение $\zeta(v)$ — булевское.

Если ζ — означивание переменных в S, то пара $B = (S, \zeta)$ называется *расширенным состоянием*. Пусть State(B) = S, Assign(B) = ζ, Voc(B) = Voc(A), Var(B) = Dom(ζ). Если Dom(ζ) — пуста, то S и (S, ζ) не различаются.

Пусть $B = (S, \zeta)$ — некоторое расширенное состояние, $\overline{v}$ и $\overline{a}$ — кортежи размерности n переменных и элементов состояния S соответственно. Пусть ζ' — новое означивание переменных в S, полученное из ζ присваиванием (назначением или переназначением) переменным v_i значений a_i для всех $i = 1, \ldots, n$ соответственно. Тогда для функции ζ' используется обозначение $\zeta(\overline{v} \mapsto \overline{a})$, а расширенное состояние (S, ζ') обозначается B($\overline{v} \mapsto \overline{a}$).

Вычисление термов

Расширенное состояние S и терм t называются *подходящими* друг другу, если словарь S содержит все функторы t, а множество переменных S включает все переменные t.

Пусть S и t подходят друг другу. Значение $\mathrm{Val}_S(t)$ терма t в состоянии S определяется индуктивно. Если t — переменная и $\zeta = \mathrm{Assign}(S)$, то $\mathrm{Val}_S(t) = \zeta(t)$. Если $t = f(\overline{u}) = f(u_1,\ldots,u_n)$, то

$$\mathrm{Val}_S(t) = f_S(\mathrm{Val}_S(\overline{u})),$$

где $\mathrm{Val}_S(\overline{u})$ обозначает последовательность $\mathrm{Val}_S(u_1)$, …,$\mathrm{Val}_S(u_n)$.

Если это не вызывает неоднозначного толкования, запись $\mathrm{Val}_S(t)$ можно сокращать до $\mathrm{Val}(t)$.

Некоторые терминологические замечания. Если вычисление терма t порождает значение **true** в состоянии S, то в этом случае говориться, что t *выполняется* в S и что S *удовлетворяет* t. Если вычисление t порождает **false**, то говориться, что t *не выполняется* в S.

Правила перехода и программы

В этом разделе описываются правила, по которым организуется переход процесса, моделируемого ASM, из одного состояния в другое.

Развитие вычислительного процесса характеризуется изменением различных параметров состояний ASM: изменением значений функций и содержимого суперуниверсума и отдельных универсумов, добавлением и удалением базисных функций. Простейшим, атомарным изменением состояния вычислений мы будем считать изменение значения одной из функций в одной точке. Этот прием позволяет описать любые необходимые нам изменения состояний.

Так, добавление элементов к суперуниверсуму можно обойти с помощью универсума *Reserve*. Для расширения некоторого универсума U за счет нового элемента поместим некоторый резервный элемент a в U. Реализовать это можно, присвоив новые значения U и *Reserve*: $U(a) :=$ **true** и $Reserve(a) :=$ **false**.

Для удаления (добавления) некоторого элемента a из универсума (в универсум) U достаточно установить $U(a) =$ **false** (**true**).

Что касается добавления или удаления базисных функций, то этих операций в принципе можно избежать за счет подходящей формализации данной абстрактной машины. Предположим, что некоторый алгоритм реализуется программой, которая

время от времени создает новые функции. На новые функции можно смотреть как на данные. Например, можно ввести универсум ***F*** всех одноместных операций на некотором универсуме ***U***. Далее, можно определить двуместную базисную (универсальную) функцию ***Apply***, такую, что

$$\forall f \in F \; \forall u \in U \; Apply(f, u) = f(u).$$

При необходимости объявить функцию с произвольным числом аргументов можно применить обычный трюк — использовать функции одного, но спискового аргумента.

Таким образом, базисный язык ASM не допускает расширений или сокращений сигнатуры или суперуниверсума.

Правила перехода

Команды или инструкции определяемого языка спецификаций называются ***правилами перехода*** или просто ***правилами***. Единственным простейшим правилом является правило ***локального изменения функции*** или ***замещения*** или просто ***замещение*** состояния *S*:

$$f(\bar{t}) := t_0, \qquad (1)$$

где f — (нестатический) функтор некоторой базисной функции, все элементы кортежа $\bar{t}$ и t_0 — основные термы и все функторы правила принадлежат словарю состояния *S*.

Смысл данного правила заключается в следующем. Сначала одновременно вычисляются t_0 и все термы кортежа $\bar{t}$, т.е. $\mathrm{Val}_S(t_0)$ и $\mathrm{Val}_S(\bar{t})$. Затем значение $f_S(\mathrm{Val}_S(\bar{t}))$ устанавливается в $\mathrm{Val}_S(t_0)$. Это значит, что если f_S не имела значения в точке $\mathrm{Val}_S(\bar{t})$, то она его приобретет, иначе старое значение будет заменено новым. В любом случае этим значением станет $\mathrm{Val}_S(t_0)$.

Следующее правило, несколько более сложное, — ***охраняемое*** (***условное***) ***замещение***:

if *b* ***then*** *u* ***endif*** или ***fi*** // формально *b* = **true** (2)

где *b* — ***охраняющее условие*** — булевский терм, и *u* — замещение.

Смысл правила: если b принимает значение **true**, то выполняется u, иначе ничего не делается. Все эти вычисления выполняются в одном состоянии машины.

В принципе условное замещение является избыточной формой правила. Достаточно ввести логическую функцию $Cond(x,y,z)$, интерпретируемую в любом состоянии следующим образом:

$$Cond(x,y,z) = \begin{cases} y, \text{ при } x = true, \\ z, \text{ при } x \neq true. \end{cases}$$

Тогда приведенное выше условное замещение вида (2) будет иметь тот же эффект, что и безусловное вида (1):

$$f(\bar{t}) := Cond(b, t_0, f(\bar{t}))$$

Однако считается, что последняя запись менее наглядна (но более точна).

Еще одна допустимая форма условного замещения имеет вид

if b ***then*** u_1 ***else*** u_2 ***fi*** (3)

Последняя форма по определению эквивалентна множеству из двух условных замещений вида (2):

if b ***then*** u_1 ***fi***

if $\neg b$ ***then*** u_2 ***fi*** // $b \neq$ **true**

что в свою очередь является частным случаем правила вида

R_1
R_2
...
R_n (3')

— произвольного конечного множества замещений (здесь все R_i — простые или условные замещения).

Интерпретатор абстрактной машины выполняет все замещения из множества замещений (условных или безусловных) одновременно (в оригинале --- параллельно), так что порядок их выполнения несуществен. Например, правило

$a := f(a)$
$b := f(a)$

устанавливает a и b в одно и то же значение.

Для того чтобы лучше понять, как выполняются множества правил, можно представить себе ***демона*** (интерпретатор), который вычисляет все подлежащие вычислению в данном состоянии термы, а затем выполняет все необходимые изменения.

В целом описанный порядок вычислений сохраняется и в том случае, когда отдельные замещения противоречат друг другу. Например, если выполняется множество правил

a := **true**

b := **false**

в состоянии, в котором a и b принимают одно и то же значение, или, что еще нагляднее, выполняется множество

a := **true**

a := **false**

то тот же демон **недетерминировано** выбирает одно из двух противоречащих друг другу замещений и выполняет его. Так, в нашем примере a примет одно из двух истинностных значений.

Теоретически множество правил можно было бы интерпретировать как последовательность. Такой подход легко снимает проблему противоречивых правил, кроме того, он более естественно подходит для конструирования макросов.

Несмотря на то что интерпретация множеств правил как последовательностей во многих ситуациях может работать лучше, свои преимущества есть и у подхода, использующего неупорядоченные множества правил. Нетрудно заметить, что подбор подходящих условий позволяет смоделировать любую последовательность действий из множества правил. Этот подход допускает определенный параллелизм действий, что дает дополнительные удобства и позволяет наилучшим образом смоделировать заданный алгоритм. Кроме того, множественная интерпретация предоставляет естественный формализм для асинхронных распределенных вычислений (в данном контексте, однако, речь идет пока только о последовательных вычислениях).

Приведенные выше две формы правила условного замещения непосредственно обобщаются. Пусть k — натуральное число, b_0, …, b_k — булевские термы и C_0, …, C_k — конечные множества правил, тогда правилами являются и следующие две формы:

if b_0 ***then*** C_0

elseif b_1 ***then*** C_1

… (5)

elseif b_k ***then*** C_k
[***else*** C_{k+1}]
fi

Легко проверить по индукции, что обе формы правила эквивалентны множеству условных замещений. Обобщая последнее замечание, можно сказать, что любое правило или множество правил эквивалентны множеству условных замещений.

Программы

Программа ASM **—** это конечное множество правил перехода.

Выполнение программы осуществляет ***интерпретатор*** абстрактной машины (демон). Выполнение начинается в одном из (любом) состояний машины, которое объявляется ***начальным***.

Завершение работы машины происходит в ***заключительном*** состоянии. Заключительным считается любое состояние, в котором выполняется ***условие завершения*** **—** любое фиксированное до начала работы машины условие вида Val($t_1 = t_2$) = **true** (в более короткой форме **—** $t_1 = t_2$), где t_1 и t_2 **—** основные термы. Таким образом, можно считать, что все правила перехода данной машины имеют охраняющее условие $t_1 = t_2$.

Пример. Стековая машина

Приведем пример ASM, моделирующей работу стека. Построим машину, вычисляющую выражение, заданное в обратной польской записи (RPN). Например, выражение

$$(1 + 2) \times (5 + 6)$$

в RPN имеет вид $1\ 2 + 5\ 6 + \times$

Это выражение должно стандартным образом интерпретироваться стековым механизмом.

В рассматриваемом примере речь может идти не только об арифметических операциях над множеством целых чисел. Пусть имеется непустое множество *Data* и непустое множество *Oper* всюду определенных (для простоты) бинарных операций над *Data*. Например, *Data* может быть множеством целых, а *Oper* может быть {+, ×}. Предположим, что выражение в RPN задается в виде списка, элементами которого являются операнды (данные) или операторы.

Стековая машина считывает из входного файла список по одному элементу. Если считывается элемент данных, то он помещается в стек. Если считывается знак операции, то из стека извлекаются два элемента данных, к ним применяется соответствующая операция, а затем результат помещается в стек. В начале процесса

стек пуст. Для примера, приведенного выше (при условии обычной интерпретации операторов), стек последовательно проходит следующие состояния: (), (1), (2 1), (3), (5 3), (6 5 3), (11 3), (33).

Для того чтобы построить требуемую абстрактную машину, введем в рассмотрение необходимые множества и функции. Применяемые в этом примере имена функций и переменных легко выявляются из контекста.

Помимо уже упомянутых универсумов *Data* и *Oper* нам потребуются нульместные функции *Arg1* и *Arg2* — выделенные элементы *Data*. Для манипулирования операциями из *Oper* нужна функция *Apply*/3, такая, что *Apply*(*f*,*x*,*y*) = *f*(*x*,*y*) для всех *f* из *Oper* и всех *x*, *y* из *Data*.

Для управления вводом определяется универсум *List* всех списков, построенных из данных и операторов. Вводятся базисные функции *Head* и *Tail* над списками, наделяемые обычными свойствами. Если *L* — список, то *Head*(*L*) — первый элемент списка и *Tail*(*L*) — хвост (оставшаяся часть) списка. Выделенный элемент *EmptyList* имеет обычный смысл. *F* — выделенный (текущий) список. Сначала *F* содержит входное выражение.

Наконец, вводится универсум *Stack* всех стеков данных с обычными операциями *Push*: *Data* × *Stack* → *Stack*, *Pop*: *Stack* → *Stack*, *Top*: *Stack* → *Data*. Также вводится выделенный стек *S*. В начальном состоянии *S* пуст. Условием завершения работы машины естественно считать исчерпание входного выражения.

Поскольку стековая машина работает только с одним списком и имеет только

один стек, может возникнуть вопрос о необходимости универсумов *List* и *Stack*. Действительно, без них можно обойтись, реализуя, например, ввод и стек посредством массивов, однако это был бы слишком низкий уровень абстракции. Мы пытаемся построить ASM на уровне абстракции данного алгоритма и погрузить такие операции, как *Push* и *Pop*, в естественную для них среду.

Приведем правила перехода определяемой машины:

```
if  Data(Head(F))  then      S := Push(Head(F), S)
                             F := Tail(F)
fi
if  Oper(Head(F))  then      if      Arg1= ω     then  Arg1 := Top(S)
                                                       S := Pop(S)
                             elseif  Arg2 = ω    then  Arg2 := Top(S)
                                                       S := Pop(S)
                             else    S := Push(Apply(Head(F), Arg1, Arg2), S)
                                     F := Tail(F)
                                     Arg1 = ω
                                     Arg2 = ω
FI                  // «много» завершающих fi
```

Приведенные правила рассчитаны на то, что выражение в RPN задано корректно.

Однако, если бы исходный алгоритм включал обработку ошибок, эту обработку можно было бы формализовать.

Также можно ввести ограничение, скажем *Max*, на глубину стека. Это потребует внести незначительные изменения в описанную модель алгоритма. Для простоты можно предположить, что при попытке добавить элемент данных в стек максимальной глубины машина просто останавливается. Далее можно ввести универсум натуральных чисел с операциями $n+1$ (следующий) и $n-1$ (предыдущий) и выделенными элементами натуральных чисел *Depth* и *Max*. В начальном состоянии *Depth* = 0. Охраняющее условие первого из двух приведенных выше правил следует расширить проверкой *Depth* < *Max* и добавить присваивание *Depth* := *Depth* ± 1 там, где это необходимо.

Упражнения

Определить машину Гуревича, моделирующую:

1) очередь;
2) дерево поиска.

Статические, динамические и внешние функции

Правила перехода ASM обращаются с базисными функциями так, как это обычно принято делать в традиционном программировании: функциям назначается некоторое имя и предоставляются средства для изменений функций. В математической практике под функциями обычно понимают фиксированные отображения. Так, если значение некоторой стандартной тригонометрической функции, скажем **sin**, изменить в одной из точек ее области определения, то эта функция перестанет быть функцией **sin**. В свою очередь базисные функции — это текущие интерпретации функторов. В каждом состоянии ASM базисные функции, интерпретирующие одно и то же функциональное имя, являются конкретными, возможно различными, отображениями.

В действительности удобно проводить различие между так называемыми *статическими* и *динамическими* функциями. Это различие является синтаксическим. Динамические функции появляются в правилах перехода как объекты замещений. Статические функции не изменяются во время эволюции ASM, тогда как динамические функции могут изменяться. Так, в примере стековой машины динамическими являются только функции *S*, *F*, *Arg1* и *Arg2*. Для статических функций в словаре состояний резервируется множество статических функциональных имен.

Еще одно замечание относится к вопросу взаимодействия абстрактных машин с внешним миром. До этого момента они определялись совершенно изолированно. В то же время необходимо предоставить средства описания ввода, вывода, обработки прерываний и любого другого взаимодействия заданного алгоритма с внешним миром.

Разумеется, невозможно и бессмысленно пытаться заранее записать и сохранить в каком-то файле все мыслимые реакции внешнего мира. Предположим, что при попытке формализовать некоторый алгоритм мы обнаружили, что одна из базисных функций, скажем *f*, зависит от внешней среды. Пусть реализация данного алгоритма задается компьютерной программой. Функция *f* этой программы может, например, возвращать значение нажатой на клавиатуре клавиши или отражать какую-либо реакцию операционной системы. Предположим, что функция *f* не является объектом замещения какого-либо правила перехода, т.е. имя *f* синтаксически статично. Тем не менее *f* может иметь различные значения в разных состояниях машины.

Один из способов разрешить ситуацию состоит в том, чтобы добавить еще один аргумент к функции *f*, с помощью которого можно было бы различать состояния вычислений. Например, если функция *f* читает некоторый файл (внешние данные), то позицию в этом файле можно использовать в качестве дополнительного аргумента. Такой подход создает иллюзию, что единственное, что делает программа ASM, — это изменяет свои состояния. Эта иллюзия создает значительные неудобства, и поэтому для разрешения проблемы используется более радикальный

путь.

Все функции ASM делятся на *внутренние статические* (для краткости, *статические* функции), *внутренние динамические* (*динамические* функции) и *внешние* функции. Внешние функции не могут изменяться правилами перехода, но они могут быть различными в разных состояниях машины. С точки зрения абстрактной машины внешняя функция есть *оракул*, непредсказуемый черный ящик, который используется, но не управляется машиной.

Таким образом, резюмируя, можно сказать, что работой ASM управляет воображаемый демон, выполняющий правила перехода. В случае внутренней функции f демон точно знает, как вычисляется эта функция. Например, он может иметь фиксированный алгоритм для вычисления стандартной, принимаемой по умолчанию версии f, а также иметь полную таблицу отклонений от этого стандарта. В случае внешней функции f демон располагает непредсказуемым «магическим» ящиком для вычисления f. Всякий раз, когда возникает необходимость вычисления f, демон берет подходящее число аргументов и «чудесным образом» получает результат. С точки зрения демона внешние функции являются недетерминированными.

Еще один важный вопрос касается типов базисных функций. С формальной точки зрения любая n-местная базисная функция не меняет свой тип: в любом состоянии она отображает n-ю степень суперуниверсума в суперуниверсум. Во многих случаях, однако, предпочтительнее думать в терминах многих универсумов

и, соответственно, специфицировать типы базисных функций в терминах универсумов. На самом деле мы оперируем многосортными алгебрами, хотя с технической точки зрения удобно использовать суперуниверсум.

Случай статических функций представляется самым простым — здесь тип функции является частью описания начального состояния машины. В случае динамических функций типизация играет роль *ограничением целостности*. Ограничения целостности — это утверждения (скорее в индикативном стиле чем в императивном), которые должны выполняться (удовлетворяться) в любом состоянии машины в процессе ее эволюции. Часто ограничения целостности являются неявными, хотя их можно сформулировать явным образом. Например, в стековой машине мы имеем следующие ограничения целостности: *Arg1* и *Arg2* — данные, (т.е. принадлежат *Data*), *F* — список, *S* — стек. Очевидно, эти четыре утверждения выполняются в любом состоянии машины.

Ограничения целостности могут быть сформулированы также для внешних функций. Например, часто ожидают, что значения внешних функций имеют значения определенных типов.

Если ограничения целостности нарушаются, то идеальная машина должна остановиться.

Ограничения целостности не обязательно должны составлять часть описания ASM. Необходимый эффект действия ограничений целостности может быть достигнут использованием дополнительных правил. Однако на практике

ограничения целостности могут быть удобным способом избежать утомительного написания избыточных правил. Например, принятие ограничения, устанавливающего, что некоторая внешняя функция производит значения определенного типа, дает возможность обойти написание правил, описывающих реакцию на нарушение этого ограничения.

Пример. Машина Тьюринга

Приведем описание ASM, моделирующей машину Тьюринга.

Введем три непустых конечных универсума ASM — *Control*, *Char* и *Displacement*, универсум $FinalState \subseteq Control$. Пусть *InitialState* и *CurrentState* — выделенные элементы *Control*, *Blank* — выделенный элемент *Char*.

Как обычно, некоторые вводимые нами функции будут статическими, некоторые динамическими; в необходимых случаях мы будем специально это оговаривать. Так, *CurrentState* — динамический выделенный элемент (динамическая нульместная функция, значение которой может изменяться от состояния к состоянию); в начальном состоянии машины $CurrentState = InitialState$.

Еще один универсум, *Tape* — бесконечное счетное множество; его элементы называются *ячейками*. *Head* — динамический выделенный элемент. *Move* — (возможно, частичная) функция: $Tape \times Displacement \to Tape$.

Существует много различных модификаций машин Тьюринга: одно- и многоленточные, ленты могут быть бесконечными в обе стороны или полубесконечными (односторонними) и т.д. Разновидность нашей машины Тьюринга определяется ограничениями, вводимыми на *Displacement* и *Move*. Например, можно выбрать универсум $Displacemrnt = \{+1, -1\}$, и функцию *Move*,

такую, что универсум *Tape* с одноместными операциями *Move*(c, +1) и *Move*(c, –1) изоморфен множеству натуральных чисел с операцией *следующий* и частичной операцией *предыдущий*. В этом случае мы получаем модель с линейной полубесконечной лентой. Аналогично можно выбрать *Displacement* и *Move* для модели с двумя линейными лентами или одной двумерной и т.д.

Определим несколько дополнительных (динамических) функций: *TapeCont* отображает ячейки ленты в символы: *Tape* $\rightarrow$ *Char*, *NewState*, *NewChar* и *Shift* отображают декартово произведение *Control* $\times$ *Char* в *Control*, *Char* и *Displacement* соответственно. Потребуем, чтобы условие *TapeCont*(*c*) = *Blank* выполнялось для всех ячеек *c*, за исключением, быть может, конечного подмножества.

Определим правила перехода машины:

CurrentState := *NewState*(*CurrentState, TapeCont*(*Head*))

TapeCont(*Head*) := *NewChar*(*CurrentState, TapeCont*(*Head*))

Head := *Move*(*Head,Shift*(*CurrentState, TapeCont*(*Head*))

Как обычно, на каждом шаге работы машины все правила выполняются одновременно. Условие завершения работы машины — *CurerntState* $\in$ *FinalState*.

Обычное определение машины Тьюринга, которое можно встретить в учебниках, на первый взгляд выглядит короче. Например, машина Тьюринга — это пятерка объектов, включающая множество состояний, множество символов, множество

правил перехода, начальное состояние и множество заключительных состояний. Однако одного этого определения недостаточно для определения семантики машины Тьюринга. Из него не следует наличие ленты, оно не содержит описание конфигурации машины и т.д. В свою очередь ASM, моделирующая машину Тьюринга, включает в себя все необходимые, в том числе упомянутые дополнительные определения. Что касается короткого определения, то его можно считать удобной математической нотацией для определения машин Тьюринга.

Перечислим ограничения целостности: $CurrentState \in Control$, $Head \in Tape$, все значения $TapeCont \in Char$. Все эти ограничения легко проверяются и могут быть представлены в форме $t = \mathbf{true}$, где t — основной терм.

Отметим, что ограничение на значения функции *TapeCont* имеют вид универсального утверждения или утверждения с квантором всеобщности: $\forall\ x \in Tape\ (TapeCont(x) \in Char)$. Оно может быть обосновано с помощью индуктивных рассуждений: полагая, что это свойство изначально выполняется, сводим его к проверке ограничений вида $Char(TapeCont(Head)) = \mathbf{true}$.

Разумеется, далеко не всегда легко проверить нарушение условия вида $t = \mathbf{true}$. Более того, в общем случае эта проблема неразрешима. Чтобы это показать, модифицируем пример с машиной Тьюринга и введем универсум $Nonhalting \subseteq Control$. Очевидно, *проблема остановки* или *завершения* работы машины Тьюринга сводится к проблеме нарушения ограничения $CurrentState \in Nonhalting$.

Пример с универсумом *Nonhalting* является довольно искусственным. На практике ограничения чаще всего являются проверяемыми, и демон ASM, способный вычислить ограниченное число термов одновременно, способен в каждом состоянии проверить, нарушено ли ограничение вида $t = \mathbf{true}$.

Имеется еще одна тонкость, связанная с проблемой завершения работы машины. Проблема остановки легко сводится к следующей *проблеме совместности* (множества правил): по данной ASM определить, встретится ли когда-нибудь во время ее работы состояние с взаимно противоречивыми замещениями. Следовательно, проблема совместности ASM неразрешима.

Можно предположить, что в определенных приложениях могут использоваться только совместные ASM. К счастью, довольно часто для обеспечения совместности правил оказывается достаточно незначительных синтаксических ограничений. Иногда удобно потребовать, чтобы для каждой базисной функции f охраняющие условия различных замещений f взаимно исключали друг друга. Это гарантирует, что никакая базисная функция не изменяется дважды на одном шаге. (Поскольку проблема взаимного исключения данного набора булевских формул является алгоритмически сложной, а именно она ко-NP полна, можно потребовать некоторого варианта явного формулирования взаимного исключения.) В общем случае каждая машина A может быть преобразована в совместную машину A', делающую два шага на каждый шаг машины A и завершающую работу всякий раз, когда A встречается с условием несовместности (по желанию при этом может

выставляться флаг ошибки в значение **true**). Идея, таким образом, заключается в проверке совместности правил A в данном состоянии A до выполнения правил.

В приведенной модели машины Тьюринга использовалась бесконечная лента. Однако определение можно перестроить так, чтобы применялась конечная лента, растущая по мере возникновения потребности в новых ячейках. Для этого можно воспользоваться (динамическим) универсумом *Reserve* — счетным подмножеством суперуниверсума, и внешним выделенным элементом *New*, для которого формулируется ограничение целостности $New \in Reserve$ (можно также определить вариант и с внутренней функцией *New*). Всякий раз, когда возникает необходимость в добавлении к ленте новой ячейки, элемент *New* удаляется из *Reserve* и добавляется к ленте. Когда элемент *New* вычисляется в следующий раз, он снова принадлежит *Reserve*. Естественно, добавление нового элемента к ленте требует некоторых действий, технические детали которых оставляются для упражнения.

Упражнения

1. Перестроить определение модели машины Тьюринга так, чтобы использовалась конечная растущая лента, а источником новых ячеек служил счетный динамический универсум *Reserve*.
2. Построить модель конечного автомата.

Последовательные и непоследовательные машины

Последовательная ASM — это, в сущности, конечное множество условных замещений функций. Иногда последовательные машины отождествляют с программами, и для этого имеется достаточно оснований. Однако бывает удобно не приводить формальное определение. Так, довольно часто точно определяется язык теории первого порядка, множество теорем и т.д., но собственно понятие теории первого порядка оставляется неформальным.

Подобное более широкое толкование понятие ASM также может дать свои преимущества. Конструирование абстрактной машины для некоторого заданного алгоритма обычно не ограничивается только написанием программ. Например, необходимо описать класс статических структур, которые предположительно могут потребоваться для представления всех возможных состояний алгоритма, или класс начальных статических структур, как, например, это было сделано нами при определении машин Тьюринга. Все это может оказаться удобным считать частью определения ASM.

Приведенные выше определения базисных правил последовательных машин не могли содержать (свободных) переменных. Тем не менее возможно обобщение базисных правил, снимающее это ограничение. Такое обобщение возможно даже в случае последовательных машин (как синтаксический сахар) для того, чтобы

сделать тексты программ более краткими, однако в случае неограниченного параллелизма это уже необходимо.

П р и м е р

Рассмотрим некоторый алгоритм обхода дерева, который всякий раз, встречая зеленую вершину, красит всех ее потомков в красный цвет. Это можно выразить правилом

if $Color(C) = green \ \ and \ \ x \in Child(C)$ ***then*** $Color(x) := red$ ***fi***

которое не эквивалентно никакому конечному множеству базисных правил. В подобных ситуациях иногда используются ограниченные кванторы всеобщности [12].

Для того чтобы продемонстрировать, почему может возникнуть необходимость в применении кванторов всеобщности, рассмотрим еще один **пример**:

Предположим, что упомянутый выше алгоритм обхода дерева красит текущий узел C в зеленый цвет всякий раз, когда все потомки C — красные:

if $(\forall\, x \in Child(C))\ (Color(x) = red)$ ***then*** $Color(x) := green$ ***fi***

Еще один упомянутый, но не рассмотренный нами вопрос касается **асинхронных распределенных вычислений**.

Понятие последовательной ASM базируется на правилах перехода из одного состояния в другое. В случае асинхронных вычислений необходимо уточнить понятие состояния. Синтаксически абстрактная машина для распределенных вычислений может выглядеть как машина, которая работает в (дискретном) последовательном времени. Различие проявляется в том, как эти машины работают. Вместо одного демона (интерпретатора) можно ввести группу демонов, отвечающих за выполнение различных правил. Всякий раз, когда охраняющие условия оказываются истинными, демоны выполняют требуемые изменения. Демоны могут работать как обычно, выполняя свои действия мгновенно, либо использовать принцип задержанных вычислений («ленивые» демоны). Примеры применения ASM для моделирования асинхронных вычислений можно найти в работах, посвященным таким языкам программирования как, например, Оккам [12] и Парлог [4].

Еще одно направление развития понятия ASM — композиция абстрактных машин. Познакомиться с этим вопросом можно в [6].

Большой пример. Семантика языка Си

Основываясь на понятии ASM, рассмотрим формальное определение операционной семантики языка программирования Си (в определении Кернигана и Ричи). Мы представим лишь фрагмент семантики языка, поскольку полное описание не вписывается в рамки небольшого примера. За полным описанием рекомендуем обратиться к [10]; по той же причине мы не будем здесь заострять внимание на особенностях той или иной реализации языка.

Так как здесь ставится задача определить именно семантику, а не синтаксис языка, мы предположим, что вся синтаксическая информация, относящаяся к заданной программе (на языке Си), известна до начала процесса вычислений и зафиксирована в статических функциях.

ASM позволяют представить семантическую спецификацию языка на нескольких различных уровнях абстракции. Имея несколько подобных машин, можно исследовать семантику отдельных возможностей языка программирования на желаемом уровне абстракции, опуская при этом несущественные детали, а также облегчить понимание семантики в целом.

Количество уровней абстракции и степень детализации могут варьироваться в зависимости от цели исследования. Мы предложим четыре абстрактные машины, последовательно уточняющие друг друга. Последняя машина (нижнего уровня)

описывает все детали языка Си.

Перечислим уровни абстракции, соответствующие определяемым машинам.

1. Операторы (условные, цикла и т.д.), т.е. уровень управления.
2. Выражения (уровень вычислений).
3. Размещение и инициализация памяти.
4. Вызов функций и возврат управления.

Что касается обработки ошибок (например деления на ноль или разименования указателей на некорректный адрес), то она в сильной степени зависит от реализации. Даже то, что именно является ошибкой, также зависит от реализации. Ясно, что нетрудно предусмотреть и формализовать подходящую реакцию системы на нестандартную ситуацию, однако этот вопрос мы здесь не рассматриваем.

В нашем фрагменте описания семантики языка мы рассмотрим только первую машину, моделирующую самый высокий уровень абстракции — модель структуры управления языка Си.

Некоторые базисные функции

Универсум *tasks* состоит из элементов (имен), представляющих задачи, выполняемые программным интерпретатором. Понятие задачи является весьма общим, например, задачей может быть выполнение оператора, инициализация переменной, вычисление выражения и т.д. Элементы этого универсума определяются конкретной Си-программой, подлежащей выполнению абстрактной машиной. Выполняемые задачи часто сопровождаются признаками, характеризующими специфику задачи; указанные признаки составляют универсум *tags*.

Выделенный элемент *CurTask*: *tasks* указывает текущую задачу. Порядок выполнения задач определяется статической (!) функцией *NextTask*: *tasks* $\rightarrow$ *tasks*, указывающей следующую выполняемую задачу после завершения текущей (имеются и другие функции, управляющие порядком выполнения задач). Статическая функция *TaskType*: *tasks* $\rightarrow$ *tags* указывает на тип действия, выполняемого задачей.

Универсум *results* содержит значения, которые могут появиться в качестве результатов вычислений.

Классификация операторов языка Си

Правила перехода абстрактной машины данного уровня абстракции определяются структурой множества операторов языка. Имеются шесть категорий операторов языка Си.

1. Операторы-выражения, вычисляющие составляющие их выражения.
2. Операторы выбора или ветвления (*if*, *switch*).
3. Операторы цикла (*for*, *while*, *do*…*while*).
4. Операторы передачи управления (*goto*, *continue*, *break*, *return*).
5. Операторы с метками (операторы *case* и *default*, используемые в области действия оператора *switch*, и оператор-обьект оператора *goto*).
6. Составные операторы.

Операторы-выражения

Операторы-выражения имеют одну двух форм:

оператор-выражение ::= ; // …;;… пустое выр.

выражение ; // ...;2+2;…;x;…

Выполнение оператора-выражения сводится к вычислению составляющего его выражения (если оно не пусто). Эта операция может сопровождаться побочными эффектами (например присваиванием значений переменным), а также может не заканчиваться. В абстрактной машине данного уровня вычисление выражений управляется внешней функцией *TestValue*: *tasks* → *results*.

Поскольку данный оператор помимо вычисления выражений не выполняет никаких дополнительных функций, машина просто переходит к выполнению следующей задачи. Соответствующее правило перехода:

if *TaskType*(*CurTask*) = *expression* ***then*** *CurTask* := *NextTask*(*CurTask*) ***fi***

где признак *expression* — один из элементов универсума *tags*.

Операторы выбора

Условный оператор *if* имеет одну из следующих форм:

условный_оператор ::= *if* (*выражение*) *оператор1*

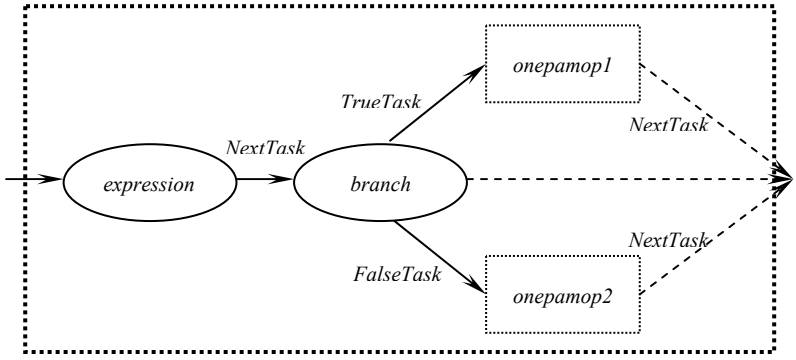
if (*выражение*) *оператор1 else оператор2*

Выполнение оператора *if* начинается с вычисления условия. Если результат не равен нулю, выполняется *оператор1*. Если результат равен нулю и оператор *if* содержит выражение *else*, выполняется *оператор2*, в противном случае выполняется оператор, следующий за оператором *if*. Статические (частичные) функции **True***Task*: *tasks* → *tasks* и **False***Task*: *tasks* → *tasks* указывают задачу, которая должна выполняться в том случае, когда условие оператора *if* принимает ненулевое или нулевое значение соответственно. Решение о выборе направления, принимаемое оператором *if*, связывается с некоторым элементом универсума *tasks*, имеющим признак *branch* (другими словами, для этого элемента значение функция *TaskType* = *branch*).

Представленный ниже рисунок иллюстрирует типичную для оператора *if* схему управления. На рисунке изображен граф задач, где овалы представляют задачи, помеченные дуги — соответствующие меткам одноместные функции, прямоугольники — подграфы, стрелки — поток управления. Если оператор *if* не содержит выражение *else*, то подграф *оператор2* вместе с входящей и выходящей

дугами, показанный в нижней части рисунка, заменяется на изображенную пунктиром дугу **False***Task*, соединяющую задачу *branch* с задачей, следующей за оператором *if*.

Типичная схема управления оператора *if*



(Если подграфы *onepamop1* или *onepamop2* содержат оператор передачи управления, то изображенная схема может измениться: *NextTask* может указывать на задачу, которая не является непосредственно следующей за оператором *if*.)

Правила перехода для задачи *branch*:

if *TaskType*(*CurTask*) = *branch* ***then***

if *TestValue*(*CurTask*) ≠ *0* ***then*** *CurTask* := **True***Task*(*CurTask*)

elseif *TestValue*(*CurTask*) = *0* ***then*** *Curtask* := **False***Task*(*CurTask*)

FI

Второй оператор ветвления — *switch* — имеет в некотором смысле более простую семантику, чем оператор *if*.

Синтаксическая форма оператора *switch*:

оператор_switch ::= *switch* (*выражение*) *тело*

где *тело* — это оператор, обычно составной.

Выполнение оператора *switch* сводится к вычислению *выражения*-условия и передаче управления первому вложенному в данный оператор *switch* оператору *case*, помеченному константой, совпадающей со значением условия, или к оператору *default*. Если указанные операторы не будут найдены, управление передается оператору, непосредственно следующему за оператором *switch*. Константы, помечающие операторы *case*, должны быть уникальными в пределах одного оператора *switch*, количество операторов *default* не превосходит единицы. Таким образом, можно определить статическую (!) частичную функцию *SwitchTask*: $tasks \times results \rightarrow tasks$, указывающую на следующую выполняемую задачу для данного значения условия. Соответственно, правила перехода из состояния выполнения задачи с признаком *switch* к следующей задаче непосредственно определяются функцией *SwitchTask*.

Упражнение. Определить правила перехода оператора switch.

Операторы цикла

Синтаксическая форма оператора *while*:

оператор_while ::= *while* (*выражение*) *тело*

Выполнение оператора сводится к циклическому вычислению *выражения*-условия с последующим выполнением *тела* оператора, пока значение условия не равно нулю. Для реализации этой схемы управления достаточно использовать уже определенные выше типы задач: *expression* и *branch*, следовательно, достаточно и уже определенных правил перехода.

Упражнения

1. Изобразить типичную схему управления оператора *while* с использованием типов задач *expression* и *branch.*
2. Определить формальную семантику операторов *do…while* и *for.*

Операторы передачи управления

Приведем без комментариев правила перехода для данной категории операторов:

if *TaskType*(*CurTask*) = *jump* ***then*** *CurTask* := *NextTask* ***fi***

Упражнения

1. Обосновать приведенные правила перехода для всех типов операторов передачи управления языка Си.
2. Определить семантику операторов с метками и составных операторов языка Си.

Начальное и заключительное состояния

В начальном состоянии машины верхнего уровня абстракции функция *CurTask* указывает на первый оператор программы.

Заключительным состоянием машины данного уровня является любое состояние, в котором CurTask = **undef**. *В этом состоянии значение* TaskType*(***undef***) =* **undef** *(и не выполняются никакие правила).*

Литература

1. *А.И.Мальцев*. Алгоритмы и рекурсивные функции. М., 1965.
2. *E.Börger, S.Mazzanti*. A Practical Method for Rigorously Controllable Hardware Design // The Z Formal Specification Notation. LNCS 1212 / Springer, 1997. P. 151–187.
3. *E.Börger, E.Riccobene*. A Mathematical Model of Concurrent Prolog. CSTR-92-15. Bristol, 1992.
4. *E.Börger, E.Riccobene*. A formal specification of Parlog. TR-1/93. Pisa, 1993.
5. *E.Börger, D.Rosenzweig*. A Simple Mathematical Model for Full Prolog. TR-33/92. Pisa, 1992.
6. *P.Glavan, D.Rosenzweig*. Communicating Evolving Algebras: Part 1. Manuscript. Zagreb, 1992.
7. *Yu.Gurevich*. Logic and the challenge of computer science // Current Trends in Theoretical Computer Science / Ed. E.Börger. Computer Science Press, 1988. P. 1–57.
8. *Yu.Gurevich*. Evolving Algebras: An Attempt to Discover Semantics // Current Trends in Theoretical Computer Science / Eds. G. Rozenberg and A. Salomaa. World Scientific, 1993. P. 266-292.

9. *Yu.Gurevich*. May 1997 Draft of the ASM Guide. CSE-TR-336-97. Michigan, 1997.

10. *Yu.Gurevich, J.Huggins*. The Semantics of the C Programming Language // Computer Science Logics. LNCS 702 / Springer, 1993. P. 274–308.

11. *Yu.Gurevich, J.Morris*. Algebraic operational semantics and Modula-2 // 1st Workshop on Computer Science Logic. LNCS 329 / Springer, 1988. P. 81–101.

12. *Yu.Gurevich, L.S.Moss*. Algebraic Operational Semantics and Occam // 3rd Workshop on Computer Science Logic. LNCS 440 / Springer, 1990. P. 176–192.

13. *G.Gottlob, G.Kappel*, *M.Schrefl*. Semantics of Object-Oriented Data Models — The Evolving Algebra Approach // Next Generation Information Technology. LNCS 504 / Springer, 1991 P. 144–160.

14. *J.K.Huggins*. Kermit: Specification and Verification // Specification and Validation Methods / Ed. E.Börger. Oxford, 1995. P. 247–293.

15. *C.Wallace*. The Semantics of the C++ Programming Language // Specification and Validation Methods / Ed. E.Börger. Oxford , 1995. P. 131–164.

16. *C.Wallace*. The Semantics of the Java Programming Language: Preliminary Version. CSE-TR-355-97. Michigan, 1997.

СОДЕРЖАНИЕ