Денис Бь0рнер: Что такое тип?

Перевод: А.Г. Пискунов

17 октября 2011 г.

АННОТАЦИЯ

Документ содержит некоторые выдержки из монографии 'Абстрагирование и моделирование' Дениса Бь0рнера посвященные обсуждению понятия тип. Перевод является существенным дополнением к обзору [2].

Содержание

1 ВВЕДЕНИЕ

Текст документа является переводом раздела 5 Types, подраздела 6.5 Type Definitions и, частично, 18 Types in RSL первого тома монографии Дениса Бь0рнера (Dines Bj0rner) Software Engineering 1. Abstraction and Modelling (см. http://www2.imm.dtu.dk/~db/Software_Engineering). Текст, не являющийся переводом, выделяется как замечание.

С языком спецификаций RSL можно познакомится в [3, 1, 15].

2 ТИПЫ

Понятие типа является, возможно, наиболее крупным вкладом, который программирование внесла в математику. Понятие типа оказалось почти настолько же всепроникающим, как и понятия размерности и измерений в физике.

Определение

В первом приближении, под типом мы будем понимать именованное множество величин.

Упрощенно говоря, типы считаются множествами величин. Значениями таких множеств, то есть его элементами, могут быть логические, численные, множества, прямые произведения, функции, отношения, списки и отображения, где элементы составных типов (множеств, прямых произведений, функций, отношений, списков и отображений) сами могут состоять из значений.

Замечание

Похоже что в первоисточнике было пропущено слово элементы или значения (в английском тексте вставка в квадратных скобках не Бь0рнера):

Types are, simplifying, taken to be sets of values. The values of type sets, i.e., their elements, are such as Booleans, numbers, sets, Cartesians, functions, relations, lists and maps where the [values of] composite types (sets, Cartesians, functions, relations, lists and maps) themselves consists of values.

В этом разделе читатель коротко познакомится с фундаментальным понятием типа. Разработчик программного обеспечения постоянно мыслит в терминах типов. Таким образом, как само понятие типа, так и абстрактное и конкретное владение им критичны для разработчика.

Этот раздел поверхностен. Будет введено понятие типа. В главах 2 - 4 приводятся примеры различных типы, а в главах 6 - 9, равно как и в главах 10, 13-16 вводится понятие типа. Понятия типа языка RSL суммируется в главе 18. После чего оно будет использоваться до конца монографии. Таким образом с данного вводного раздела мы начинаем длинное путешествие в, возможно, наиболее важную концепцию программирования: теорию типов и её практическое использование.

Мир полон явлений - сущностей, на которые можно указывать. Некоторые сущности имеют общие свойства и оказываются 'одного и того же типа', другие - нет, и оказываются 'разных типов'. В некотором абстрактном смысле, это понятие была вначале введено философами, позже - математиками и, намного позже появилась в языках программирования чтобы справиться с 'похожестью' , соответственно, с 'различностью'.

От читателя ожидается некоторое базовое знакомство с элементарными сторонами понятия типа некоторых языков программирования. Рассматривая и анализируя конкретные примеры понятия типа в языках программирования можно очень быстро получить базовое представление о более абстрактном понятии типа.

В этой секции вводится базовое понятие на котором позже будем основывать дальнейшие идеи о типах. Этой основой будут являться: сорты (то есть, абстрактные типы), сорт конкретные типы, атомарные типы, имена типов, выражения типов, конструкторы типов и факт, что типы и значения образуют дополняющие понятия.

2.1 Типы и величины

Как можно обьяснить понятие типа? Например, следующим образом: пусть существует такое явление как человек ростом 1 метр 79 сантиметров, 67 лет отроду и очень тяжелый. Эти три измерения будут указывать на такой отдельный феномен как сущность. Конкретный человек есть сущность описываемая, то есть, характеризуемая только что указанными тремя атрибутами или признаками сущности. С первого взгяда понятно,

что атрибуты представляют, то есть, характеризуют величины и, со второго взгляда, эти атрибуты разных типов: рост, возраст и вес. Таким образом, значение сущности может быть либо атомарного либо составного типа.

Значение сущности - человек имеет составной тип и типа включают типы роста, возраста и возраста как компоненты, которые имеют атомарные тип, то есть, их нельзя разложить на более мелкие составляющие. Некоторые сущности имеют постоянные значения, другие - переменные значения. День рождения человека определенно фиксирован. Пол человека - обычно фиксирован. Возраст человека изменяется все время.

Сущности редко меняют свой тип. Редко обсуждаемый пример сущности, который может рассматриваться как изменяющий тип, заключается в следующем: Вначале, некоторая сущность может рассматриваться как деревянное кресло. То есть, нечто полезное и используемое для сидения (utility). Затем кресло меняет свой тип так, что становится антикварiатом, его можно показывать, нл нельзя на нем сидеть, нельзя использовать. С нашей точки зрения оно перестает быть мебелью, используемой для сидения. Или оно может сломаться и становится кучей дерева и, с этого момента, может трактоваться как горючий материал для плиты. То есть, его можно использовать, но совсем в другом смысле! Моделирование типов - включая изменение типов - часто называется как моделирование данных. Другими словами: типы и величины идут рука об руку.

Далее, бдует сказано гораздо больше про понятие типа, понятия атрибута и значения, так же как про использование этих понятий в моделировании мира, моделировании требования к програмному обеспечению и в выражении программной реализаций этих моделей.

2.2 Явление и понятие типа

2.2.1 Явления и понятия

Определение.

Под явлением будем понимать некоторую, физически проявляемую вещь, нечто такое, на что можно указывать или измерять при помощи физических инструментов.

Любой конкретный человек является таким явлением.

Определение.

Под понятием будем понимать абстракцию, нечто в нашем уме.

Понятия обычно обобщают классы связаных явлений.

Следуя ранее изложенному, классы подобных явлений или подобных понятий в типы.

Эта секция посвящена изучению отношения между явлениями, понятиями и типами.

2.2.2 Сущность: неделимая и составная

Определение.

Под сущностью будем понимать представление явления или понятия.

Определение.

В общих чертах, под представлением будем понимать способ разговаривать о чем нибудь, способ записать что либо.

Представление явления не является явлением, это только наш способ обсуждать его. Еще раз с другой стороны: представление явления, до тех пор, пока оно не содержится внутри компьютера, будет рассматриваться как информация. Если представление содержится внутри компьютера, то это представление будет рассматриваться как данные. Данные это формализованное представление информации.

Определение.

Под неделимой сущностью будем понимать сущность, которая не состоит из некоторых подсущностей.

Человек может рассматриваться как неделимая сущность в том смысле, что его голова, руки, ноги и так далее с некоторой точки зрения не могут считаться сущностями. Возможно они рассматриваются как сущности с точки зрения хирурга, но такой взгляд может быть нежелательным с точки зрения не хирурга, которяй не составляет, как в механической инженерии, человека из головы, одной левой руки и так далее!

Пожалуйста, запомните, что это Вы тот, кто решает надо ли считать некоторое явление (или понятие) неделимым или нет.

Определение.

Под составной сущностью будем понимать сущность про которую можно сказать, что она составлена из других подсущностей.

Машину можно рассматривать как составную сущность в смысле, что можно считать, что она составлена из мотора, трансмиссии, левой передней двери и так далее, в то время, когда эти подсущности рассматриваются как сущности, с помощью которых можно собрать машину.

2.2.3 Атрибуты и значения

Определение.

Под атрибутом будем понимать именованное свойство некоторого типа, причем у другой сущности так же названное свойство может иметь такой же или другой тип.

2.2.3.1 Атрибуты и значения неделимых сущностей Неделимая сущность может иметь несколько атрибутов.

Человек, считающийся неделимой сущностью имеет кроме других атрибутов: имя (фиксированное значение, скажем, Денис Бь0рнер), рост (может изменяться, скажем, 1979 см), пол (фиксированное значение, мужской), и так далее.

Таким образом 'полное значение' неделимой сущности может быть составным значением.

2.2.3.2 Атрибуты и значения составных сущностей Можно считать, что способ которым была составлена составная сущность является атрибутом и этот атрибут отличается от множества атрибутов соответствующих подсущностей.

2.2.3.3 Пример. Сеть путей (Road Net): сущности и атрибуты: Сеть составлена из множества сегментов (дорог) и множества коннекторов. Сегменты не содержат коннекторы, а дотрагиваются до двух. Сегмент - это сущность. Коннектор не содержит сегментов, а соединяет один или больше сегментов (один, если дорога является тупиком). Коннектор - это сущность. Каждый сегмент имеет атрибуты: уникальный идентификатор,

название дороги, длину сегмента, кривизну сегмента, его покрытие (гудрон или др.). Никакой из перечисленных атрибутов не является сущностью. Коннектор имеет следующие атрибуты: идентификатор, название, множество идентификаторов соединяемых сегментов. Никакой из атрибутов не является сущностью. Сама сеть путей имеет атрибут с информацией как она составлена из ее подсущностей.

Определение.

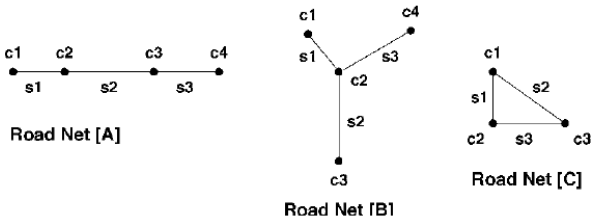
Атрибуты составной сущности: надо различать атрибуты подсущностей и атрибут составной сущности: Пусть составная сущность состоит из сущностей $c_1, c_2, ..., c_n$. Каждая из индивидуальных подсущностей c_i, где $i = 1...m$, имеет атрибуты $C_{i_1}, C_{i_2}, ..., C_{i_m}$. Кроме этого, составная сущность имеет атрибуты C. Последний атрибут обьявляет каким образом составляется сущность, то есть, как мы решили ее составить. Например: в C записано, что c это последовательность компонет c_i, или что c это множество компонено c_k, или что c состоит из компоненты c_{l_p}, соседней с компонентой c_{l_q} соседней с компонентой .... соседней с компонентой c_{l_r}.

Позже способ составления будет выражаться операторами типа. То есть, операторами, которые определяют как типы компонент образуют результирующий тип.

Определение.

Значения составной сущности: С каждым атрибутом связывается текущее значение. Пусть составная сущность c состоит из сущностей $c_1, c_2, ..., c_m$. Каждая индивидуальная сущность $c_i, i = 1...m$ имеет текущие значения $v_{c_{i_1}}, v_{c_{i_2}}, ..., v_{c_{i_m}}$. Дополнительно, текущее значение атрибута C сущности c обозначим v_c, Полное текущее значение для c составляется, как предписано в C, из полных текущих значений подсущностей: $v_{c_{i_1}}, v_{c_{i_2}}, ..., v_{c_{i_n}}$.

2.2.3.4 Пример. Значения сети путей: Продолжаем пример 2.2.3.3 . Конкретные сети составлены из трех сегментов как показано на рисунке 2.2.3.4 , см. подфигуры [A]-[C]. Соседние сегменты соединяются коннектором. На подфигурах [A] и [B] отображены два и три тупика.



Представления трех различных значений сети путей.

Полные значения сетей отличаются, в основном, по свойствам их топологии. Все три сегмента могут иметь одинаковые значения, то есть, одинаковую длину, идентификатор, имя, покрытие, и так далее. Но, как можно заметить, рассматривая рис. 2.2.3.4 существует различие в значениях самих сетей.

Мы попытались, немного не формально, получить некоторые идеи про неделимые и составные сущности, их атрибуты и значения. Эти идеи должны быть изложены более точно. И это является основной причиной текущего тома!

2.2.3.5 Обсуждение Возможно записать сеть путей в терминах графов начальной математики:

$G : (S, C, K)$

где S - обозначение для множества сегментов. Например,

$$\{$$

$\mathrm{s}_1, s_2, s_3\}$

$$.$$

$\mathrm{C}-.,\{\mathrm{c}_1, c_2, c_3, c_4\}$

$$-[A][B],$$

$\{c_1, c_2, c_3\}$

$$[C].$$

K−., [s$_1$ ↦{c_1, c_2}
]$[A]$.2.2.3.4 ., (,).

Представление, ориентированное на свойства алгебраического типа и аналитические функции:

```
type
  G0, S, C
value
   obs_Ss: G0 -> S-set
  ,obs_Cs: G0 -> C-set
  ,obs_K: G0 -> (C -m->(S-m->C))
```

вернуло бы для значения сети g0, соответствующей подфигуре [A]:

```
Obs_Ss (g0) = {s1, s2, s3}
Obs_Cs (g0) = {c1, c2, c3, c4}
Obs_Ks (g0) = [c1 +> {s1},
               c2 +> {s1, s2},
               c3 +> {s2,s3},
               c4 +> {s3}]
```

Спецификация ориентированная на множества, прямые произведения и отображения:

```
type
  G1 = (C >< S >< C )-set
  G2 = C -m-> ( S -m-> C)
```

содержит следующие значения g1 и g2:

```
g1:{(c1,s1,c2), (c2,s1,c1), (c2,s2,c3), (c3,s2,c2), (c3,s3,c4), (c4,s3,c3)}
g2: [ c1 +> [s1 +> c2],
      c2 +> [s1 +> c1, s2 +> c3],
      c3 +> [s2 +> c2, s3 +> c4],
      c4 +> [s3 +> c3]]
```

То есть, особенности обьявления типов опубликованные в этой работе заменяют обычный способ, которым математики определяют математические структуры. Наши особенности определения типа связаны со свойствами определения функций и допускают определение очень сложных и разнообразных математических структур для сущностей и типов.

2.3 Понятия типа в языках программирования

Рассмотрим некоторые обычные понятия языков программирования.

2.3.1 Некоторые примеры

Из классических языков программирования, таких как Algol 60, Pascal, C, C++ и Java, хорошо известно понятие типа, похожее на уже изложенное.

2.3.2 Простые типы

Три синтаксические конструкции после ключевого слова var:

```
[1]  var i integer,
[2]      b Boolean,
[3]      c character;
```

требуют выделение памяти для трех переменных:

- первая - для i - чтобы иметь достаточно места для хранения целых величин, например, между -2^n и $2^n - 1$ (для некоторых n, таких как 16, 32 или 64), где n это размер в битах ячейки памяти (такие ячейки также называются, байт, слово или двойное слово);
- вторая - для b - чтобы иметь достаточно места для хранения логических величин, то есть, $true$ или $false$;
- И последняя для - чтобы иметь достаточно места (скажем, один байт или одно слово) для хранения символьных величин.

Можно указать несколько особенностей, полезных для понимания приведенного выше пример:

- используется ключевое слово var (обьявление переменной) чтобы указать обьявление переменной;
- записано три обьявления;
- каждое из обьявлений имеет два части: имя переменной (i, b, c) и встроенное имя типа ($integer$, $Boolean$, $character$);

- для каждой переменной неявно задается конкретное представление в памяти;
- и то, что имя переменной (очень вероятно, что уникальное) указывает на место в оперативной памяти, которое может содержать величины соответствующего типа.

2.3.3 Составные типы:

```
[4]  type r =
[5]      record ( i integer,
[6]               b Boolean,
[7]               a array [1..m,1..n]  of char);
[8]  var p r;
```

Как уже выше обьяснялось, в строке [8] находится обьявление переменной, однако имя переменной p связывается с типом r, который определяется в строках [4]-[7] (и не является, как ранее, встроенным именем типа). Это определенное имя является сокращением тексту, который записан с права от знака '=' в четвертой строке, то есть, строчки [5-7]. В этих строчках можно видеть, что определен тип запись (record), которая составляется из значений других типов. Запись содержит три именованных поля, которым соответствуют некоторые участки памяти для содержания целых и логических величин, равно как матрицу из m строк и n столбцов, каждый элемент которой предназначен для хранения символьной величины. Эта матрица является примером массива с переменными границами ([1..m,1..n]). Между прочим, имена полей выбраны похожими на имена ранее введенных переменных специально, чтобы запутать читателя. Возвращаясь к переменной p ([8]): это сущность (переменная) типа r, которая (кроме того, что она является переменной) имеет частичные атрибуты: целый, логический и символьный.

2.3.4 Присваивание и проверка типа выражения

```
[9]   i   := i+1;
[10]  b   := (if  i > p.i then true else false end);
[11]  p.i := p.i + i;
[12]  c   := p.a [i,p.i]
```

Последний пример не совсем связано с нашей главной целью. Нашей целью было введение такого же понятия типа, как и в классических

языках программирования. Вместо этого текущий пример иллюстрирует такие понятия императивных языков программирования как присваивание, выбор значения из поля записи и выражения. Строка [9] содержит простое присваивание: Значение целой переменной i увеличивается на единицу. В строке [10] записано еще одно присваивание, условное выражение и выбор значения из поля записи: в логическую переменную b присваивается значение $true$, если значение i больше чем значение, содержащееся в поле i записи p, иначе в b присваивается значение $false$. И, наконец, строка [12] содержит, выглядящее несколько хитроумным, индексирование массива: символьной переменной c присваивается элемент символьного массива один индекс которого берется из простой переменной i, а второй - из поля i записи p.

2.3.5 Обсуждение

До сих пор в связи с типами уже было использовано два вида ключевых слов: имена типов и конструкторы типов. Имена типа подобно $integer$, $Boolean$, $character$ обозначают типы величин конкретного вида (denote types of specific kinds of values). Конструкторы типов дают возможность, подобно $record$ и $array$, совместно с разделителями, идентификаторами и именами типов формировать или конструировать новые, определяемые пользователем, типы. Эти использование ключевых слов сохраняется также для функций высшего порядка. Таким образом, имена типов - это идентификаторы и могут быть либо встроенными ключевыми словами (подобно $integer$, $Boolean$, $character$), либо определяемыми пользователем (подобно r) типами; и составными выражениями типа такими как $record(id1te1, id2te2, ..., idnte$ где idj и tej обозначают идентификатор поля записи и выражение типа, соответственно. Ясно, что имя типа это наиболее простое выражение типа. Кроме того, из примеров выше видно, что определение типа состоит из пары имя типа (как r) и выражения типа (как $record(id1te1, id2te2, ..., idnten)$).

Будем говорить, что когда имя типа r определено конкретно, то дается некоторая модель. Модель, данная для r, состоит в том, как запись располагается в памяти компьютера: именованные, последовательно расположенные участки памяти. Позже станет видно, что не все имена типа обязаны иметь конкретную модель.

Конструктор, формирующий запись, выглядит подобно:

```
record ( * *, * *, ..., * *)
```

где, первая звездочка в каждой паре * * мыслится как места для различных идентификаторов селекторов поля записи, а вторая - как места для, не обязательно различных, имен типов (в более общем случае - выражений типа).

Конструктор, формирующий массив, выглядит подобно:

```
array [*..*, *..*, ..., *..*] of *
```

где первая и, соответственно, вторая звездочка это места, для целочисленных выражений, которые задают нижнюю и верхнюю границу каждой размерности массива, в то время как последняя звездочка после ключевого слова *of* - место для выражения типа.

Если не учитывать выделение памяти с его естественными ограничениями на расположение элементов (или полей) составного типа, оба понятия - и типы и величины могут быть найдены в почти любом языке абстрактных спецификаций, и, соответственно, в RSL. Поэтому будем полагать вышеприведенные примеры случаями конкретных структур данных, в то время, как сначала должны быть смоделированы (спецификациями домена и требованиями к алгоритмам (in domain specifications and in requirements prescriptions)) информационные структуры. Данные считаются представлением информации в компьютере. Таким образом, спецификации доменов и требования к алгоритмам дают возможность абстрагироваться от представления памяти. С другой стороны, будут широко использоваться и типы, и имена переменных для типов (даже без уточнения являются ли эти переменные, переменными аппликативного или функционального программирования).

2.4 Абстрактные типы или сорты

Вернемся к сути понятия типа, но не в языках программирования, а в языках спецификаций. Большинство языков спецификаций имеют встроенные типы, такие как целые, логические и символы. Они обычно используются для неделимых типов, то есть, для типов, значения которых не могут быть разложены на составляющие (так, что бы это поддавалось осмысленной интрепретации с точки зрения прикладной области). Некоторые, в основном, модель - ориентированные, языки спецификаций, например, RSL, VDM-SL, Z предлагают конструкторы типа подобные записям и массивам, что бы строить составные типы из других, уже существующих или ранее

определенных типов. Далее в этой секции познакомимся с конструктором прямого произведения.

Встречаются также языки спецификаций - алгебраические Cafe-0BJ, CASL, позволяющие вводить абстрактные типы или сорты. Сорт это тип у которого нет явно присвоенной модели (о котором ничего не известно):

```
type
 A, B, C
```

Типы A, B и были названы, но не было сделано никаких определений.

До этого момента был немного использован синтаксис языка RSL: ключевое слово *type* сообщает читателю, что сейчас последуют (перед другими словами) декларации типа. Приведенная выше декларация ввела имена A, B, C как имена типов. Про типы A, B, C можно думать что они являются некоторым множеством значений типа A, B, C.

Позже, в ходе рассуждений, может оказаться является ли сорт неделимым типом или составным. В последнем случае его величины можно будет анализировать при помощи составных частей (то есть, величин) с соответствующими типами.

2.5 Встроенные и конкретные типы

Продолжим пошаговое знакомство с уже частично известными типами RSL. Сейчас будем продолжать думать о типе, как о возможно бесконечном множестве, чем то похожих друг на друга величин.

Добавим некоторый синтакс, чтобы можно было именовать и определеять типы:

```
[0] type
[1]   I = Int, B = Bool, C = Char
[2]   P, Q, R
[3]   K = P >< Q >< R
```

Int, $Bool$, $Char$ это ключевые слова языка RSL. Они обозначают множества целых, логических и символов соответственно. P, Q, R имена типов определенные пользователем. Они обозначают абстрактные типы - сорты. K - имя определенного пользователем типа и обозначает прямое произведение, то есть, множество троек из значений соответствующих сортов. Следующий текст на RSL

```
[4]  value
[5]    p0, p1, ..., px :P, q, q1,..., qn : Q, r,r1, ..., ry: R
```

обозначает связывание. Например, идентификаторы $p0$, $p1$, ..., px - все различны обозначают неопределенные значения типа P.

Связывание величин в строке [10]:

```
[6]   type
[7]     A, B
[8]     L =  A >< b >< ... >< C
[9]   value
[10]    (a0,b0, ...,c0), (a1,b1, ...,c1), ...,  (aX,bX, ...,cX):L
```

связывает свободные и различные имена $a0, b0, ..., c0$, $a1, b1, ..., c1$, ..., , $aX, bX, ..., cX$ со произвольными значениями соответствующих типов. Типы K и L используются для значений из прямых произведений.

Прокоментируем немного только что введенные элементы языка RSL. Ключевое слово *type* (из строк [0,6]) сообщает о том, что за ним последуют имена или определения типов. В строке [1] после слова *type* показаны три определения конкретных типов; в строке [2] приводятся определения абстрактных типов; в строке [3] опять приводится определение конкретного типа. Причем первые три определения просто дают другие имена (а именно I, B, C) уже существующим типам. Последнее определение конкретного типа дает имя K прямому произведению типов $P >< Q >< R$. Инфиксный символ $><$ похож на конструктор типа записи record (, , ...,). Таким образом, $><$ это конструктор прямого произведения, равно как и в строке [8].

Ключевое слово *value* (см. строчки [4,6]) указывает, что $p0, p1, ..., px$ различные имена произвольных (не обзязательно различных) значений типа P, и так далее.

Составное связывание в строке [10] - выражает, что индивидуальные значения $a, b, ..., c$ с различными индексами в тексте $(a0, b0, ..., c0), (a1, b1, ..., c1), ..., (aX, bX, ..., c$
сгруппированы в кортежи.

2.6 Проверка типа

Идея ассоциировать типы с идентификаторами полезна вдвойне: во первых, читателя информируется о специальном использовании идентификатора

и, в тоже время, синтаксический анализатор получает возможность выявления некорректного использования типизированных других идентификаторов. Кратко обсудим последнее утверждение.

2.6.1 Типизированные переменные и выражения

Давайте рассмотрим следущий сегмент программы из примера 2.3.3

```
[1]  var i integer := 7,
[2]     b Boolean   := true,
[3]      c character := 'd';
...
[4]  type r =
[5]      record ( i integer,
[6]               b Boolean,
[7]               a array [1..4,1..2]  of char);
[8]  var p r;
...
[9]   i   := i+1;
[10]  b   := (if  i> p.i thdn true else false end);
[11]  p.i := p.i + i;
[12]  c   := p.a [i,p.i]
```

Выражения и операторы выглядят вполне нормально.

2.6.2 Ошибки связанные с типизацией

Если бы в строке [9] было бы написано $b * 7$, или в строке [10] - $if b > p.a$, или в строке [11] - $p.i := c$, тогда можно было бы считать, что чтото не правильно.

Что же может быть не правильно? В строке [9] (при условии записи $b * 7$) - b уже объявлена как логического типа, а операция умножения для него не задана. В строке [10] (при $if b > p.a$) b является логической величиной, а $p.a$ обьявлено как символ и нет операции сравнения логической и символа, В строке [11] ($p.i := c$), $p.i$ была обьявлено целой, а c - символом и нет возможности присваивать символьные значения целым переменным.

2.6.2.1 Правильно сформированная сеть путей Продолжим разбор сети путей из 2.2.3.3 . Можно указать два вида ограничений к типу для которых должна проверяться сеть дорог:

- если под сетью путей будем понимать такую сеть, которая не содержит изолированных друг от друга дорог, тогда надо уточнить требования к примеру:
  - от каждый коннектора сети можно доехать до любого другого коннектора сети.
  - или сеть не должна распадаться на два или более изолированных подграфов. Изоляция понимается в том смысле, что каждый сегмент дороги является двусторонним.
- если под сетью будем понимать такую сеть, в которой каждый не тупиковый сегмент может быть как односторонним, так и двусторонним путем. Тогда, чтобы гарантировать отсутствие изолированности, характеристика примера +ex5.2 должна быть уточнена следующим способом:
  - любой тупик является двусторонним сегментом.
  - любой сегмент является или односторонним, или двусторонним сегментом.
  - от любого коннектора можно доехать до любого другого коннектора, двигаясь по разрешенному направлению связанных сегментов (Односторонний сегмент имеет только одно разрешенное направление).

2.6.3 Обнаружение ошибок типизации

Имея обьявленные переменные с типами, можно делать выводы о том, выглядит ли корректно каждая операция, индексирование и присвоение. Такая деятельность называется проверкой типизации. Гораздо позже будет уточнено что это значит 'быть такого-то типа' и как можно использовать это знание. Формализация помогает автоматизацию проверки типизации.

Много научных сотрудников и инженеров полагают, что понятие типа используется только из за проверки типизации. Наш взгляд несколько шире: проверка типизации важна для как можно более раннего обнаружения ошибок спецификации. Хотя абстрагирование в терминах сортов и конкретных типов тоже важно, так как это дисциплинирует ум.

2.7 Типы как множества, типы как решетки

До сих пор типы трактовались как множества значений. Такая трактовка очень часто оказывается полезной, но не всегда. Если предполагается, что тип D будет включать пространство функций из D в D, то теоретико - множественная трактовка оказывается не удовлетворительной. Оказывается просто невозможным обьяснить значение равенства

$D = D \to D$

Что бы разрешать такое равенство, необходимо ввести, например, порядок на элементы множества типа (что называется доменом типа). Такая теория типов будет только упомянута. Это теория типов в том смысле, что она дает возможность разрешать произвольные равенства типов. То есть, умение придавать подходящее значение типам рефлексивных функций является критерием информатики. Дана Скотт основал теорию типов в только что подсказанном смысле([7, 8, 9, 10, 12]. Для введения в теорию типов можно почитать [16, 17, 13], а также [6] или его перевод на русский - [4]).

2.8 Резюме

Закончено начальное обсуждение понятия типа из языка RSL. Для этого познакомились с базовыми, встроенными типами (Int, $Bool$, $Char$), причем все они являются конкретными и неделимыми типами. Были введены абстрактные типы, то есть, сорты, а так же конкретные, но составные типы, которые образуются декартовым произведением, при помощи конструктора типа $><$. Далее (в частности, в разделе 3.1.2) будут введены другие аспекты понятия типа из RSL.

3 ФУНКЦИИ

Понятие функции - математическое понятие. Это понятие, наряду с типами, имеет первоочередную важность. Невозможно наблюдать функцию, математические функции можно 'наблюдать' путем ее применения к аргументам и и получения результатов.

Определение.

Под функцией понимается математическая сущность, которая может быть применена к аргументам (некоторым сущностям) что бы получить результат применения, то есть, возвращаемые значение.

3.1 Алгебра функций

Алгебра состоит из множества значений и множества операций. К этой паре можно добавить название для алгебры. В этой секции эти вопросы будут обсуждены в другом порядке: значения, название алгебры, операции.

3.1.1 Функции

Значениями алгебры функций является пространство всех функций этой алгебры. Функция - это такая мистическая вещь, которую можно применить к аргументам из ее множества определения и получить результат из ее множества значений. Нельзя увидеть, собственно, функцию точно также как нельзя увидеть просто число. Это математические сущности, которые характеризуются их свойствами.

3.1.2 Типы функций

Во первых, обсудим способ записывания выражений типа для обозначения функциональных пространств, затем будем записывать типы функций высшего порядка. Есть синтаксические отличия между всюду определенными, $->$ и частичными, $-\not>$ функциями:

```
Выражение типа:                 Определение типа
                                type
 A -> B                           TF = A -> B
 A -~-> B                         PF = A -~-> B
```

Такие выражения типа это составные имена (то есть, сигнатуры) алгебры функций. Имена типа TF и PF это простые имена алгебры функций. Текст $f = A -\not> B$ означает, что записан тип функции f.

Таким образом, символ $->$ это функция инфиксного конструктора типа: она получает два аргумента - два типа (то есть, два множества значений) A и B и возвращает пространство всех полностью определенных функций из множества определения A во множество значений B. A, $-\not>$ это функция другого инфиксного конструктора типа: она получает

два аргумента - два типа (то есть, два множества значений) A и B и возвращает пространство всех частично определенных функций из множества A во множество B. То есть, во множестве A существуют значения для которых во множестве $A \xrightarrow{\sim} B$ найдутся функции не определенные на этих значения.

С точки зрения синтаксиса, $->$ равно как и $\xrightarrow{\sim}$ это инфиксные операторы. Их операнды должны быть выражениями типа.

3.1.3 Типы функций высшего порядка

В приведенных выражениях типы A и B, сами могут быть функциональными:

```
type
  A = P -> Q
  B = U -> V
  F = (P -> Q)  -> (U -> V) is A -> B
```

Вообще, говоря выражения типа:

```
A -> B -> C is A -> (B -> C) ~= (A -> B) -> C
```

То есть, инфиксный констуктор типов функций ассоциативен справа. Операторы *is* иΛ= только что использовались в металингвинистическом смысле: они выглядят как операторы RSL, но на деле ими не являются. В RSL нельзя сравнивать типы, поэтому здесь эти операторы надо понимать как математические.

3.1.4 Недетерминистические функции

Пусть функции f определена как:

```
value
  m,n: Nat
  f: Nat -~-> Nat ,
  f(i)  is
    let j:Nat :- j > i in i+j end
  ... f(7) ... f(9) ... f(13) ...
  g: Real -> Nat ,
  g(j) is  m
  ...
 g(1/if n = 0 then 1000000000000 else n end) ... g(1/(1+n))
```

где $Real$ и Nat обозначают вещественные, соответственно, натуральные числа. Говорят, что функция f называется недетерминистической. То есть, она возвращает произвольное натуральное число, но не обязательно то же самое для следующего вызова f. Недетерминистические функции из типа A в тип B тоже имеют сигнатуру частичных функций: $A -\!\!\!\!\!\!\sim > B$.

3.1.5 Функции - константы

Определенная выше функция g является функцией - константой. Ее определение использует недетерминистическое определение m; m это любое натуральное число, но оно определено только один раз. Таким образом g - это функция константа. К какому бы аргументв ее не применяли, она возвращает одно и тоже значение. В частности, следующее определение:

```
type
  A
value
  a: A
  f: Unit -> A, f() is a
```

подсказывает, что значения любого типа могут рассматриваться как функции - константы:

```
value
   zero, one, two : Unit -> Nat;
   tt, ff : Unit -> Bool;
   zero() is 0,
   one () is 1,
   two () is 2,
   tt() is true,
   ff() is false
```

Замечание

Символ $Unit$ обозначает значение (). Он используется тогда, когда определяется функция без аргументов. Вызов такой функции без аргументов f записывается как $f()$.

3.2 Заключение

Закончено знакомство с сущностью функций: они отображают аргументы из множества определения в (то есть, возвращают) результаты множества значений, их можно определять, именовать, использовать и абстрагировать (использовать лямбда - операцию). Кроме того, выяснено, что функции имеют тип - отображение из множества определения во множества значения. Вместе с именем функции эта тройка называется сигнатурой функции.
Далее, функции могут быть либо всюду определенными либо частично определенными. Далее, будут введены понятия сюрьективных, иньективных и биективных функций.

4 ТИПЫ В RSL

Замечание

> Типы языка RSL, несомненно, представляют интерес, с той точки зрения, что этот язык спецификаций является достаточно удачным и применяемым инструментом. Набор его конструкций, а значит и набор его типов и конструкторов для создания новых типов, позволяет адекватно описывать проектируемые программные продукты. Каждый конструктор типа является сокращенной формой для введения данного типа через конструкции языка RSL, в основном, через список аксиом. Что и будет проиллюстрировано ниже на примере перечислений и рекурсивных типов.

Как говорилось выше в RSL есть абстрактные (сорты) и конкретные типы. Конкретные типы делятся на

- $Bool$;
- Int;
- Nat;
- $Real$;
- $Char$;

- *Text*;
- множества;
- списки;
- декартовы произведения;
- отображения;
- функции;
- обьединения;
- записи (рекурсивные и не рекурсивные);
- перечисления;
- явно выделяемые подтипы.

Замечание

Причем перечисления и записи - являются специальными случаями операции явного выделения подтипа. По сути, берется сорт к которому дописываются аксиомы, необходимые чтобы получить требуемые свойства для перечисления или записи.

4.1 Перечисления

Перечисления используются, если необходимо иметь конечное число различаемых, неделимых сущностей без какого либо описания.

4.1.0.1 Пример. Игральные карты: Набор из 52 двух игральных карт без джокера обычно моделируется как:

```
type
  Suit = = club | diamond | heart | spade
  Face = = ace | two | three | ... | ten | knight | dame | king
  Card = Suit x Face
```

4.1.1 Общая теория

Под перечислением понимаются неделимые, явно выписанные значения. Пусть t и $t^{'}$ два символа из перечисления. Тогда либо $t = t^{'}$ либо $t \neq t^{'}$. Равенство и неравенство это операции определенные на символах перечисления. Обобщенный пример выглядит так:

```
type
 Token == token_1 | token_2 | ... | token_n
```

Он определяет n неделимых величин: $token\_1, token\_2, ..., token\_n$. Символ == сигнализирует, что будет использоваться конструктор перечисления - |.

Приведенное выше определение является сокращением следующего полного определения:

```
type
  Token
  value
    token_1: Token,
    token_2: Token,
    ...,
    token_n: Token
  axiom
  [ disjointness of enumerated tokens ]
    token_1 ~= token_2 /\ .../\ token_1 ~= token_n  /\
    token_2 ~= token_2 /\ .../\ token_2 ~= token_n  /\
    ...,
    token_n-i  ~=  token_n
```

Такое перечисление используется вместе с аксиомой индукции (то есть, считается что она выполняется):

```
axiom
  [ enumerated token induction ]
  V p:Token -> Bool :-
  p(token_1) /\ p(token_2)/\ ... /\ p(token_n) => all token:Token :- p(token)
```

Для любого предиката p, если предикат выполняется для каждого символа из перечисления, то p выполняется для всех символов типа данного перечисления. Таким образом, если в качестве p взять

```
value
  p: Token -> Bool
  p is -\ t:Token :- t=token_1 \/ t=token_2 \/ ... \/ t=token_n
```

то можно утверждать, что нет никаких символов типа $Token$, кроме $token\_1, token\_2, ..., toke\_n$

Замечание

С нашей точки зрения важно то, что в расширенном описании не используется никаких символов, кроме символов логики и принадлежности типу. Тип $Token$, вообще говоря, задается списком аксиом.

- $Token$ - это аксиома существует тип $Token$;
- $token\_1 : Token$ это аксиома - существует $token\_1$ символ типа $Token$.

4.2 Выделение подтипа

Выражение для выделения типа выглядит следующим образом:

```
{| b: B:-  P(b)|}
```

обычно оно встречается в связи с определенниями типа:

```
type
   A = {|b: B:-  P(b)|}
```

Оно означает, что тип A содержит те значения из типа B, которые удовлетворяют предикату P.

4.3 On Recursive Type Definitions

We refer to Sects. 13.5.1, 14.5.1, 15.5.1, 16.5.1 and 17.4.1 for discussions on the issue of defining types recursively. Dana Scott provided the basic research that now serves as the theoretical setting for our understanding of types: [7, 8, 9, 10, 11, 12].

5 ЗАКЛЮЧЕНИЕ

Дальнейший текст не является переводом.

Итак, формальный ответ на вопрос: 'Что такое тип?' в узком смысле звучит так: 'это то, что перечислено в разделе 4 '. И эти типы, по видимому, позволяют строить любое программное обеспечение.

Далее, рассмотрим двукратное выделение подтипа

```
type
   C,
   B = {|c: C:-  Q(c)|},
   A = {|b: B:-  P(b)|}
```

по видимому, альтернативное определение типа A

```
type
   A = {|c: C:-  Q(c)  /\  P(c) |}
```

определяет то же множество значений, на которых выполняются предикаты Q и P. Во вторых, любой из встроенных типов ($Bool$, Nat, Int, ...) может быть явно выделен из некототого сорта со специально подобранным предикатом (который, в свою очередь, может трактоваться как набор аксиом связанных операцией и: $/\backslash$). В разделе 7.14 монографии [14] можно посмотреть пример введения натуральных чисел через аксиоматику Пеано.

Таким образом, на вопрос: 'Что такое тип?' можно вполне ответить в смысле работы [2, 5]:

Тип - это название и список аксиом, которым удовлетворяют значения типа, то есть, тип - это спецификация.

Предметный указатель

ССЫЛКИ

[1] Кафедра системного программирования МГУ. Методы Формальной Спецификации Программ. http://www.ispras.ru/~RedVerst/RedVerst/Lecturesandtrainingcourses/MSUcourseFormalspecificationofsoftware/RMain.html.

[2] А.Г. Пискунов, С.М.Петренко. Формализация ООП: Типы, множества и классы, 2010. http://www.realcoding.net/articles/formalizatsiya-oop-tipy-mnozhestva-i-klassy.html.

[3] А.Г. Пискунов. The RAISE Method Group: Алгебраическое проектирование класса, 2007. http://www.realcoding.net/article/view/4538.

[4] Бенджамин Пирс. Типы в языках программирования, 2010. http://newstar.rinet.ru/~goga/tapl/tapl-toc.html.

[5] А.Г. Пискунов. Об одном примере нарушения принципа подстановки Лисков, 2010. http://www.realcoding.net/articles/ob-odnom-primere-narusheniya-printsipa-podstanovki-liskov.html.

[6] B. Pierce. Types and Programming Languages, 2002. MIT Press.

[7] C. Gunter, D. Scott. Semantic Domains, 1990. [344] — v°l- B.: ed by J. Leeuwen (North-Holland, Amsterdam, 1990) pp 633-674.

[8] D. Scott. The Lattice of Flow Diagrams, 1970. pp 311-366.

[9] D. Scott. Lattice Theory, Data Types and Semantics, 1972. Symp. Formal Semantics, pp 67-106, ed by R. Rustin (Prentice Hall, 1972).

[10] D. Scott. Lattice-Theoretic Models for Various Type Free Calculi, 1973. 4th Intl. Congr. for Logic Methodology and the Philosophy of Science, Bucharest (North-Holland, Amsterdam, 1973) pp 157-187.

[11] D. Scott. Data Types as Lattices, 1976. SI AM Journal on Computer Science 5, 3 (1976) pp 522-587.

[12] D. Scott. Some Ordered Sets in Computer Science, 1982. Ordered Sets, ed by I. Rival (Reidel Publ., 1982) pp 677-718.

[13] D.A. Wolfram. The Clausal Theory of Types, March 1993. Cambridge University Press.

[14] Chris George. The RAISE Specification Language, 1999.

[15] Chris George. Introduction to RAISE. UNU-IIST report No. 249, 2002. ftp://www.iist.unu.edu/pub/techreports/report249.pdf.

[16] J.-Y. Girard, Y. Lafont, P. Taylor:. Proofs and Types, 1989. vol 7, Cambridge Tracts in Theoretical Computer Science edn (Cambridge Univ. Press, UK 1989).

[17] J.R. Hindley. Basic Simple Type Theory, October, 2002. Cambridge University Press.