МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

Е.А. Кузьменкова, А.К. Петренко

ПРАКТИКУМ ПО ФОРМАЛЬНОЙ СПЕЦИФИКАЦИИ ПРОГРАММ НА ЯЗЫКЕ RSL

МОСКВА

2008

УДК 378(075.8):004.43
ББК 32.973-018.1я73
К89

Пособие содержит справочные и методические материалы по практическому освоению основных разделов курса лекций «Формальная спецификация и верификация программ», касающихся использования языка спецификаций RSL. В пособии приводятся упражнения по основным разделам языка RSL и рекомендации по их выполнению, а также предлагаются варианты задания практикума и примеры экзаменационных билетов.

Работа над пособием выполнена в рамках образовательной программы «Формирование системы инновационного образования в МГУ».

Для студентов программистских кафедр факультета ВМиК в поддержку лекционного курса «Формальная спецификация и верификация программ» и для преподавателей, ведущих практические занятия по этому курсу.

Рецензенты:
Кузнецов С.Д., д.т.н., профессор
Соловьев С.Ю., д.ф.-м.н., профессор

Е.А. Кузьменкова, А.К. Петренко «Практикум по формальной спецификации программ на языке RSL» — М. Издательский отдел факультета ВМК МГУ (лицензия ИД № 05899 от 24.09.01), 2008. — 88 с.

Печатается по решению Редакционно- издательского Совета факультета Вычислительной математики и кибернетики МГУ ми. М.В. Ломоносова.

ISBN 978-5-89407-338-5
ISBN 978-5-317-02422-2

ВВЕДЕНИЕ

«Методы формальной спецификации программ» – завершающий курс блока «Программная инженерия». В настоящее время он состоит из трех курсов:

- Объектно-ориентированный анализ и проектирование
- Верификация программ на моделях
- Методы формальной спецификации программ

За рамками блока пока находятся вопросы организации производства программных средств (ПС), управление требованиями, верификация и валидация, оценка стоимости ПС и сроков разработки и другие. Сейчас блок представляет некоторое сужение предмета программной инженерии. В таком виде его можно было бы назвать как «Конструирование программных систем». При рассмотрении проблем конструирования программных систем необходимо не забывать о двух аспектах программной системы:

- структурном (архитектурном)
- функциональном (поведенческом)

Программисты часто увлекаются первым, тогда как для пользователя или заказчика скорее более важным представляется второй. Однако искусство разработчика программной системы состоит в том, чтобы постоянно держать в поле зрения оба ракурса рассмотрения. Поэтому для профессионального разработчика важно умение рассматривать и структурные, и функциональные аспекты системы/проекта и находить компромиссы там, где функциональные требования приходят в противоречие с архитектурными решениями.

Сопоставляя проблемы архитектурного и функционального анализа систем можно сделать вывод, что структурный аспект на практике превалирует над функциональным. Это, в частности, объясняется тем, что он опирается на достаточно хорошо известные нотации (языки программирования) и инструменты, и тем что задачи конструирования изучаются в многих учебных курсах, соответственно, специалистов в области конструирования/программирования программных систем много. Методы описания и исследования систем в функциональном аспекте не так развиты. Задачей курса «Методы формальной спецификации программ» является, хотя бы частичное, восполнение этого пробела.

Многие исследователи-специалисты по формальным методам разработки программ акцентируют свое внимание на собственно формальном аппарате, используемом для анализа спецификаций, на инструментах, которые упрощают работу с формальными спецификациями. Эти работы, конечно, важны. Но чрезмерное увлечение математической стороной дела не только не помогает в решении практических задач, но часто и строит дополнительные трудности. Здесь, как и в любой инженерной задаче, важно находить компромисс в сочетании строгих, формальных методов и инженерных, эмпирических подходов. В практике использования формальных спецификаций главной инженерной задачей является выбор метода абстракции реальной целевой системы, который позволяет построить модель поведения, которая представляет, с одной стороны, необходимое приближение поведению реальной системы, и, с другой стороны, достаточно простой, чтобы обеспечивать возможность использования выбранного формального аппарата, имеющихся инструментов для работы со спецификациями.

В полной мере формальные спецификации раскрывают свой потенциал тогда, когда они используются системно на разных или даже на всех фазах жизненного цикла программной системы. При этом на разных фазах и для разных целей могут использоваться модели различные по уровню абстракции, различающиеся аспектами, на которые они нацелены, использующие разные нотации и формальные аппараты. Соответственно, перед методами формальной спецификации стоят разные задачи: от описания эскизного замысла, до детального описания отдельных интерфейсов или протоколов и задач определения соответствия между спецификациями различных уровней абстракции, между спецификациями и реализацией.

В рамках одного курса, конечно, невозможно обхватить все методы и используемые формальные аппараты. Задача курса – показать, как формальные спецификации могут предоставить некоторую строгую концептуальную основу, позволяющую там, где это необходимо, обеспечить корректность реализации за счет доказательной демонстрации корректности самих спецификаций и соответствия реализации требованиям, заданными данными спецификациями. Авторы пособия пытались подобрать материал таким образом, чтобы студенты и преподаватели на учебных задачах смогли познакомиться с различными видами и методами использования формальных спецификаций.

Пособие состоит из трех частей. Первая часть содержит методические указания для проведения занятий практикума по освоению языка спецификаций RSL. Описываются основные понятия языка, способы спецификации функций, приводятся упражнения по основным разделам языка RSL и конкретные рекомендации по их выполнению. Вторая часть пособия содержит описание задания практикума на ЭВМ, основной целью которого является разработка спецификаций программной системы средней сложности на языке RSL, здесь же приводятся конкретные варианты заданий практикума. В третьей части содержатся примеры вариантов письменного экзамена прошлых лет с решениями.

Работа над пособием выполнена в рамках образовательной программы «Формирование системы инновационного образования в МГУ».

ГЛАВА 1. ОСНОВНЫЕ ТИПЫ ЯЗЫКА RSL. СПЕЦИФИКА RSL-ЛОГИКИ

Концепция типов в языке RSL (встроенные, абстрактные, составные). Описание подтипов. Встроенные типы языка RSL. Логика в языке RSL. Квантифицированные и условные выражения RSL. Упражнения.

Концепция типов в языке RSL

В языке RSL имеется три вида типов:

- встроенные (предопределенные),
- абстрактные,
- составные (производные).

Встроенный тип определяется набором предопределенных литералов и операций над значениями данного типа.

Абстрактный тип представлен только идентификатором соответствующего типа. Значения такого типа не уточняются, в типе отсутствуют предопределенные операции генерации значений данного типа и манипулирования с ними (кроме операций = и ~=). С помощью абстрактного типа осуществляется декларация типа, про значения которого ничего не известно, кроме возможности сравнивать их посредством операций = и ~=.

Составной тип строится из других типов посредством применения типовых операций.

Над всеми типами определены операции = и ~=.

Описание подтипов

На основе любого типа можно построить *подтип*. Для задания подтипа необходимо определить с помощью предиката некоторое ограничение на исходный (базисный) тип. В общем случае описание подтипа имеет вид:

```
ST1 = {| t : T1:- p(t) |},
```

где ST1 обозначает определяемый подтип, T1 - исходный (базисный) тип, p(t) – предикат, задающий ограничение на значения базисного типа. Значениями подтипа являются все возможные значения исходного (базисного) типа, на которых предикат-ограничение принимает значение **true**.

Встроенные типы языка RSL

В языке RSL предусмотрены следующие встроенные типы:

Bool – представляет булевские значения,
Int – целые числа,
Nat – натуральные числа,
Real – вещественные числа,
Char – символы,
Text – строки символов,

Unit – содержит единственное специальное значение **()**.

Тип **Bool** содержит два значения **true** и **false,** над значениями данного типа определены операции /\, \/, **=>**, ~, **is** (эквивалентность), =, **~=**.

Тип **Int** представляет все целые числа. Над значениями этого типа определены операции +, –, *, /, \, **, **abs**, **real**, где / означает целочисленное деление, \ – взятие остатка от деления, ** - возведение в степень, **abs** – префиксная операция для вычисления модуля числа и **real** – префиксная операция для преобразования из типа **Int** в тип **Real**. Заметим, что автоматического преобразования из типа **Int** в тип **Real** в RSL нет. Кроме того значения данного типа можно сравнивать с помощью операций <, **<=**, >, **>=**, =, **~=**.

Тип **Nat** содержит все натуральные числа 0, 1, 2, …, он является подтипом типа **Int** и задается соотношением:

Nat = {| i **: Int** :- i >= 0 |}.

К значениям этого типа применимы все операции, определенные для типа **Int**.

Тип **Real** представляет все действительные числа, к значениям этого типа применимы операции +, –, *, /, **, **<**, **<=**, >, **>=**, =, **~=**, **abs**, **int**, где **int** – префиксная операция для преобразования из типа **Real** в тип **Int**, возвращающая ближайшее по направлению к 0 целое число. Например:

int 4.6 = 4
int –4.6 = –4

Наличие десятичной точки в записи вещественных констант обязательно (1**.**0, 12**.**35 и т.д.).

Тип **Char** содержит символы, к значениям этого типа применимы операции = и ~≠, константы типа **Char** заключаются в апострофы (например: ′f′, ′Z′ и т.д.).

Тип **Text** предназначен для описания строк символов, причем каждая строка должна начинаться и заканчиваться символом ″. Например, ″ABC″, ″″ – пустая строка. К значениям этого типа применимы операции =, **~=**.

Тип **Unit** является аналогом типа **void** в языках C, C++, Java, C#, он используется в описаниях сигнатур функций для изображения отсутствия входных или выходных параметров.

Логика в языке RSL

Остановимся более подробно на вычислении логических выражений в языке RSL, поскольку принятая в языке логика имеет некоторую специфику по сравнению с классической логикой. Суть этой специфики связана с наличием в RSL специального выражения **chaos**, которое предназначено для обозначения непредсказуемого (хаотичного) поведения программы во время ее выполнения. Такое поведение может возникать в результате какого-либо отказа в программе и приводит к ситуации, когда вычисление значения какого-либо выражения может не завершиться (например, при вычислении выражения 1/x в точке x=0.0). **chaos** не принадлежит ни к одному из типов RSL и в выражениях его вхождения могут появляться в позициях, предусмотренных для значений различных типов.

Например, **chaos** может встречаться как вместо вхождения значений типа **Bool**, так и вместо типа **Int**.

Другой специфической особенностью языка RSL является принятая в нем сокращенная схема вычисления логических выражений, т.е. вычисление второго операнда выражения не производится, если значение всего выражения полностью определяется значением его первого операнда.

Ниже приводятся таблицы истинности (с учетом **chaos**) для основных логических операций. Левая колонка соответствует первому операнду, верхняя строка – второму операнду.

/\	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

\/	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

=>	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Определение операции ~ необходимо расширить соотношением:

~ **chaos** ≡ **chaos**

В таблицах истинности для операций = и **is** символы a и b обозначают выражения, значения которых отличаются друг от друга.

is	a	b	**chaos**
a	**true**	**false**	**false**
b	**false**	**true**	**false**
chaos	**false**	**false**	**true**

=	a	b	**chaos**
a	**true**	**false**	**chaos**
b	**false**	**true**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

На основании приведенных таблиц можно заметить, например, что в отличие от классической логики в RSL операции /\ и \/ не обладают свойством коммуникативности, т.е. выражения $e_1 /\backslash e_2 \equiv e_2 /\backslash e_1$ и $e_1 \backslash/ e_2 \equiv e_2 \backslash/ e_1$ уже не являются тавтологиями.

Квантифицированные и условные выражения RSL

Для обеспечения возможности альтернативного выбора при вычислении выражений в RSL введены условные выражения, имеющие следующий вид:

if expr **then** $expr_1$ **else** $expr_2$ **end,**

где expr является логическим выражением, а выражения $expr_1$ и $expr_2$ имеют один и тот же тип, который является и типом всего условного выражения. Например, вычисление модуля разности двух величин можно задать с помощью выражения:

if x > y **then** x – y **else** y – x **end**

При вычислении значений условных выражений необходимо иметь в виду следующие свойства:

if true then $expr_1$ **else** $expr_2$ **end** ≡ $expr_1$ (1),

при этом вычисление выражения $expr_2$ не производится,

if false then $expr_1$ **else** $expr_2$ **end** ≡ $expr_2$ (2),

при этом вычисление выражения $expr_1$ не производится,

if a **then** $expr_1$ **else** $expr_2$ **end** ≡
if a **then** $expr_1$[**true**/a] **else** $expr_2$[**false**/a] **end** (3),

где запись вида $expr_1$[**true**/a] обозначает подстановку в выражение $expr_1$ значения **true** вместо a.

if chaos then $expr_1$ **else** $expr_2$ **end** ≡ **chaos** (4),

при этом вычисление выражений $expr_1$ и $expr_2$ не производится.

В языке RSL допускается также более сложная форма условного выражения, имеющая следующий вид:

```
if expr1 then v_expr1
    elsif expr2 then v_expr2
        ...
          if exprn-1 then v_exprn-1
              else v_exprn end
```

Такое выражение является сокращенной записью условного выражения:

```
if expr1 then v_expr1
    else
        if expr2 then v_expr2
            else
              ...
                else
                    if exprn-1 then v_exprn-1
                        else v_exprn
                    end
                end
            ...
end
```

В качестве примера сложного условного выражения можно привести выражение, вычисляющее модуль x:

```
if x = 0 then 0
    elsif x > 0 then x
        else 0 – x end
```

Квантифицированные выражения языка RSL имеют традиционную форму и используются в основном для записи аксиом. Допускаются кванторы: **all** (∀), **exists** (∃), **exists!** (∃!).

Упражнения

1. Определить подтип:

   (a) всех положительных натуральных чисел,
   (b) всех нечетных целых чисел,
   (c) всех вещественных чисел из интервала (–1.0, 1.0).

2. Ниже приведены равенства, которые верны в классической логике. Какие из них верны в RSL?

   (a) ~(~a) **is** a
   (b) **true** \/ a **is true**
   (c) a \/ **true is true**
   (d) a **=> true is true**
   (e) a **=>** b **is** ~a ∨ b
   (f) a \/ ~a **is true**
   (g) (a /\ ~a) **is false**
   (h) (a /\ b) /\ c **is** a /\ (b /\ c)
   (i) (a \/ b) \/ c **is** a \/ (b \/ c)
   (j) (a = a) **is true**
   (k) (a **is** a) **is true**

   Указания:
   - воспользуйтесь приведенными в данной главе таблицами истинности для основных логических операций;
   - для проверки справедливости предложенных утверждений достаточно исследовать их значения на наборах данных, где хотя бы один операнд обращается в **chaos**.

   Например, для пункта (c):

   chaos \/ true is chaos,
   (**chaos is true) is false**

   Следовательно, утверждение (c) неверно в RSL.

3. Упростить следующие выражения:

   (a) **if true then false else chaos end is** ?
   (b) **if** a **then ~(a is chaos) else false end is** ?

Указание: воспользуйтесь свойствами условного выражения (1) - (4).

4. Какие из следующих квантифицированных выражений верны?

   (a) **all** i **: Int :- exists** j **: Int :-** i+j = 0
   (b) **all** i **: Int :- exists** j **: Nat :-** i+j = 0
   (c) **exists** i **: Int :- all** j **: Int :-** i+j = 0

5. Напишите RSL выражение, выражающее следующий факт:

   (a) нет наибольшего целого числа,
   (b) значение max является максимумом значений x и y,
   (c) натуральное число n является четным числом,
   (d) натуральное число n является полным квадратом,
   (e) натуральное число n является простым числом.

Указания: воспользуйтесь предопределенными операциями над встроенными типами и квантифицированными выражениями RSL.

ГЛАВА 2. ОПИСАНИЕ КОНСТАНТ И ФУНКЦИЙ

Общая структура RSL спецификации. Декартовы произведения. Выражение **let**. Характеристика основных стилей спецификации функций - явного, неявного, аксиоматического. Описание констант в различных стилях спецификации. Понятие всюду вычислимой и частично вычислимой функции. Описание функций в различных стилях спецификации. Схема описания функции. Упражнения.

Общая структура RSL спецификации

Спецификация на языке RSL состоит из набора *описаний* определенного вида. Описания верхнего уровня представляют собой *модули*. Мы ограничимся рассмотрением одного вида модулей – *схемами*, когда модуль содержит определение схемы. В этом случае описание модуля (схемы) имеет вид:

```
scheme id =
  class
        decl1
          ...
        decln
  end
```

где id – идентификатор схемы, $decl_1$,..., $decl_n$ ($n \geq 0$) – разделы описаний. Каждый раздел описаний содержит описания определенного вида и начинается с ключевого слова, задающего вид перечисленных в нем описаний. Внутри раздела все входящие в него описания перечисляются через запятую.

Кроме модулей в языке RSL имеются следующие виды описаний: *типы* (**type**), *значения* (**value**), *аксиомы* (**axiom**), *переменные* (**variable**) и *каналы* (**channel**). В самом общем случае описание схемы имеет вид:

```
scheme id =
  class
        type
          <описание типов>
        value
          <описание значений>
        axiom
          <описание аксиом>
  end
```

Декартовы произведения (products)

Декартовым произведением называется упорядоченный конечный набор значений, возможно, различных типов, например:

```
(1,2)
(1,true,"John").
```

В декартовом произведении существенен порядок следования элементов, т.е. (1,2) и (2,1) являются различными значениями.

Тип, определяющий декартово произведение нескольких типов, задается типовым выражением вида:

t_1 >< ... >< t_n, где n≥2,

и представляет собой составной тип, образованный из типов t_i ($1 \le i \le n$) путем применения типовой операции ><. Значениями такого типа являются декартовы произведения длины n (n-ки значений или кортежи) $(v_1,...,v_n)$, где каждое v_i – некоторое значение типа t_i.

Примеры записи значений декартовых произведений с указанием их типов:

(**true**, p => q) **: Bool** >< **Bool**
(x + 1,0,"this is a text") **: Int** >< **Nat** >< **Text**

Над декартовыми произведениями разрешены операции = и ~=. Допустимо также использование именования компонент декартова произведения, например, (x,y,z).

Выражение let

Выражение **let** предназначено для введения новых имен для обозначения отдельных компонент более сложного значения. При этом **let**–выражение объявляет блок, который определяет область видимости введенных в нем имен.

Наиболее часто с помощью выражения **let** производится именование компонент декартова произведения. В этом случае **let**-выражение имеет вид:

let $(x_1,...,x_n)$ = a **in**
  expr
end

где a обозначает декартово произведение, $x_1,...,x_n$ -новые вводимые имена для обозначения его отдельных компонент, expr задает выражение, в пределах которого можно использовать данные имена.

Например, пусть a и b обозначают координаты двух точек на плоскости. Тогда расстояние между двумя этими точками вычисляется с помощью выражения:

```
let (x1,y1) = a in
        let (x2,y2) = b in
                ((x2 – x1)**2.0 + (y2 – y1)**2.0)**0.5
        end
end
```

Описание констант

В RSL константа рассматривается как частный случай функции, а именно как функция без параметров с постоянным значением, поэтому для описания констант и функций в языке предусмотрен единый раздел описания значений **value**.

Описание констант в языке RSL может производиться в одном из трех стилей:

- явном (эксплицитном, explicit),
- неявном (имплицитном, implicit),
- аксиоматическом.

Явный стиль описания применяется, когда помимо типа определяемой константы непосредственно указывается ее значение. Например, RSL-спецификация вида:

```
value x : Int = 1
```

определяет целочисленную константу x=1.

Неявный стиль описания используется, если точное значение константы не указывается, а задается лишь ее тип, а также, возможно, некоторые ограничения на значение константы. Например, спецификация:

```
type Stack
value x : Int :- x > 0,
      empty : Stack
```

определяет целочисленную константу x, значением которой может являться любое целое положительное число, и константу empty абстрактного типа Stack, о значении которой вообще не сообщается никакой информации. Спецификация при этом получается неполной и может быть уточнена в дальнейшем. Такой прием называется *недоспецификацией* или *неполной спецификацией* и применяется в том случае, когда описываемое значение по каким-либо причинам не может быть определено полностью на данном этапе составления спецификации. В основном это характерно для начальных этапов разработки спецификаций.

Аксиоматический стиль описания заключается в том, что наряду с типом определяемой константы задается также некоторый набор аксиом, определяющих свойства этой константы (например, накладывающих дополнительные ограничения на ее значение). Заметим, что как для явного, так и для неявного описания можно построить эквивалентное ему описание в аксиоматическом стиле. Так, эквивалентная форма приведенных выше описаний константы x в аксиоматическом стиле выглядит следующим образом:

```
value x : Int
axiom x is 1          (для явного описания),

value x : Int
axiom x > 0           (для неявного описания).
```

Всюду вычислимые и частично вычислимые функции

Описание функции в RSL начинается с определения её сигнатуры, т.е. имени функции и типов входных и выходных параметров, здесь же задается вид функции с точки зрения возможности её вычисления для всех значений, определяемых типом входных параметров. Различаются два вида функций – всюду вычислимые и частично вычислимые функции.

Функция f, отображающая значения типа T_1 в значения типа T_2, является *всюду вычислимой* (total function), если для любого значения из T_1 f возвращает некоторое единственное значение из T_2, т.е. f обладает следующим свойством:

```
all x : T1 :- exists! y : T2 :- f(x) is y
```

Сигнатура функции f в этом случае имеет вид:

f **:** T_1 -> T_2

Всюду вычислимые функции всегда детерминированы и определены для всех значений входных параметров.

Функция f, отображающая значения типа T_1 в значения типа T_2, является *частично вычислимой* (partial function), если существует такое значение v типа T_1, для которого вычисление f(v) может либо вообще не завершиться (в этом случае f(v) **is chaos**), либо привести к недетерминированному результату, когда различные вхождения выражения f(v) могут возвращать разные значения.
Сигнатура частично вычислимой функции f имеет вид:

f **:** T_1 -~-> T_2

Частично вычислимые функции включают в себя всюду вычислимые функции.

Как правило определение частично вычислимой функции содержит описание предусловия, накладывающего некоторые ограничения на значения входных параметров и гарантирующего нормальное завершение вычисления функции при условии соблюдения этих ограничений. Предусловие задается предикатом, зависящим от входных параметров определяемой функции.

Дальнейшее определение функции подобно рассмотренному ранее определению констант может производиться в одном из следующих стилей:

- явном (эксплицитном, explicit),
- неявном (имплицитном, implicit),
- аксиоматическом.

Выбор конкретного стиля спецификации зависит от специфики решаемой задачи.

Явное описание функций

Описание функции в явном стиле (операционная спецификация) ориентировано на описание конкретного алгоритма и используется в том случае, когда явно задаётся способ преобразования входных параметров в выходные. Примером явного описания всюду вычислимой функции может служить следующий фрагмент RSL спецификации:

```
value
f : Int -> Int
f(x) is x + 1
```

В качестве примера описания частично вычислимой функции в явном стиле рассмотрим функцию p(x), вычисляющую значение 1/x:

```
value
p : Real -~-> Real
p(x) is 1.0/x
pre x ~= 0.0
```

Последняя строка данного описания содержит предусловие, представляющее собой предикат, накладывающий ограничение на значения входного параметра.

Неявное описание функций

Описание функции в неявном стиле (контрактная спецификация) нацелено на описание свойств результата функции, которые фиксируются в виде некоторого предиката в постусловии определяемой функции. Неявная спецификация, абстрагируясь от конкретного способа получения результата, во главу угла ставит формализацию отношений, связывающих между собой входные и выходные параметры, т.е. здесь описывается не сам алгоритм, а эффект его применения. Данный стиль описания соответствует более высокому уровню абстракции, чем явная спецификация, поскольку конкретный алгоритм преобразования не указывается и может быть уточнен в дальнейшем путем построения явной спецификации определяемой функции. При этом для одной и той же неявной спецификации можно предложить, вообще говоря, разные алгоритмы, эффект применения которых удовлетворяет указанной спецификации. При разработке спецификаций функций этап построения их неявных спецификаций как правило предшествует этапу построения явных спецификаций.

Ниже приведены примеры описания в неявном стиле всюду вычислимой функции f(x), возвращающей в качестве результата некоторое целое число, превосходящее входное значение:

```
value
f : Int -> Int
f(x) as r post r > x
```

и частично вычислимой функции square_root(x) для нахождения значения квадратного корня:

```
value
square_root : Real -~-> Real
square_root(x) as s
        post s * s = x /\ s >= 0.0
pre x >= 0.0
```

Причем, как видно из примеров, конкретный алгоритм получения результата остается в обоих случаях за рамками рассмотрения, описываются лишь свойства результата.

Аксиоматическое описание функций

При использовании аксиоматического стиля описания функции предлагается некоторый набор аксиом, определяющих свойства данной функции. Аксиоматическая спецификация может применяться с одинаковым успехом и для описания способа получения результата, и для описания свойств самого результата выполнения функции. Поэтому как для явного, так и для неявного описаний всегда можно предложить эквивалентное описание в аксиоматическом стиле. Кроме того данный стиль является единственно возможным при построении алгебраических спецификаций, где аксиомы описывают свойства цепочек функций и имеют специфический вид (например, f(g(x), x) **is** h(x)), т.е. в качестве аргумента функции допускается использование обращения к функции.

В качестве примеров приведем аксиоматическое описание всюду вычислимой функции f(x), эквивалентное рассмотренному выше явному описанию той же функции:

```
value
f : Int -> Int
axiom all x : Int :- f(x) is x + 1
```

и аксиоматическую спецификацию частично вычислимой функции square_root(x), эквивалентную ее неявной спецификации, также рассмотренной ранее:

```
value
square_root : Real -~-> Real
axiom
all x : Real :-x >= 0.0 =>
        exists s : Real :- square_root(x) = s /\ s * s = x /\ s >= 0.0
```

Схема определения функции

Подводя итог вышесказанному, можно предложить следующую схему определения функции:

1. Описать сигнатуру функции:
   - выбрать имя функции;
   - выбрать тип:
     (a) аргументов,
     (b) результата,
     (c) отображения:
       - всюду вычислимая функция (может быть определена для всех значений входных параметров),
       - частично вычислимая функция (необходимо предусловие);
2. выбрать стиль спецификации:
   - явный (можно задать способ вычисления результата),
   - неявный (можно описать связь входа и выхода посредством предиката),
   - аксиоматический (может быть использован всегда, необходим в алгебраических спецификациях).

Упражнения

1. Для обеспечения работы с точками на плоскости описать тип для представления:

   (a) всех точек на плоскости, принадлежащих первому квадранту и лежащих выше прямой y=x,
   (b) всех точек с целочисленными координатами, лежащих внутри круга с центром в начале координат и радиусом 5.

Указания: используйте декартово произведение и аппарат определения подтипов.

2. Используя явный стиль спецификации, определить функцию:

   (a) *is_even* для проверки заданного числа на четность,
   (b) *is_a_prime*, которая проверяет, является ли ее аргумент простым числом.

Указания: в пункте (a) можно использовать операцию \ (остаток от деления) или воспользоваться квантифицированным выражением.

3. Определить функцию *max*, возвращающую значение максимума из двух целых чисел, используя один из следующих стилей описания:

   (a) явный,
   (b) неявный,
   (c) аксиоматический.

Указания:

- воспользуйтесь приведенной в этой главе общей схемой определения функции;
- для удобства записи в пунктах (a) и (c) используйте условное выражение языка RSL.

4. Для обеспечения работы с комплексными числами описать:

   (a) тип *Complex* для представления комплексных чисел,
   (b) константу *zero* для представления комплексного числа $0 + 0i$,
   (c) константу *c*, представляющую любое комплексное число вида $x + xi$;
   (d) функции *add* и *mult* для сложения и умножения комплексных чисел,
   (e) функцию *f*, возвращающую в качестве результата некоторое комплексное число, отличное от заданного.

Указания:

- в пунктах (b) и (d) воспользуйтесь явным (explicit) стилем описания константы и функций, т.к. здесь явно указано значение константы и способ получения результатов функций по их входным значениям;
- в пунктах (c) и (e) следует использовать неявное (implicit) описание константы и функции, поскольку в условии не оговаривается явно значение константы и способ получения результата по входному значению;
- в пунктах (c) и (d) для упрощения записи удобно воспользоваться выражением **let**. С помощью этого выражения, например, для пункта (c) факт равенства действительной и мнимой частей комплексного числа *c* можно записать так:

```
let (x, y) = c in x = y end
```

5. Пусть задана следующая спецификация системы координат:

```
scheme SYSTEM_OF_COORDINATES=
  class
    type
      Position = Real >< Real
    value
```

```
        origin : Position = (0.0,0.0),
        distance : Position >< Position -> Real
        distance((x1, y1),(x2, y2)) is
                ((x2 – x1)**2.0 + (y2 – y1)**2.0)**0.5
end
```

Здесь тип *Position* предназначен для описания координат точек на плоскости, константа *origin* задает начало координат, функция *distance* определяет способ вычисления расстояния между двумя точками. Обратите внимание на то, что при описании константы *origin* и функции *distance* использован явный стиль описания, т.к. указано непосредственное значение константы и формула для вычисления расстояния.

Дополнить заданную спецификацию определением:

(a) типа *Circle* для описания окружности по ее центру и радиусу;
(b) функции *on_circle*, определяющей, лежит ли точка с заданными координатами на заданной окружности;
(c) окружности с радиусом 3.0 и центром в начале координат;
(d) константы *pos* для представления произвольной точки, лежащей на заданной окружности.

Указания:

- в пункте (a) определите вспомогательные типы *Center* и *Radius* для представления центра окружности и ее радиуса соответственно, для определения типа *Center* используйте тип *Position*, для типа *Radius* воспользуйтесь следующим описанием:

  ```
  Radius = {| r : Real :- r >= 0.0 |},
  ```

  задающим подтип действительных чисел с неотрицательными значениями;
- в пунктах (b) и (c) следует воспользоваться явным стилем описания функции и константы, можно также использовать конструкцию **let** для достижения большей наглядности записи;
- в пункте (d) используйте неявный стиль описания константы.

6. Для обеспечения работы с рациональными числами предложить спецификацию схемы *RATIONAL_NUMBERS*, содержащей определения:

   (a) типа *Rational* для представления рациональных чисел;
   (b) функции *less*, которая сравнивает два рациональных числа и возвращает **true**, если первое число меньше второго;
   (c) функции *add* для сложения двух рациональных чисел;
   (d) функции *max* для определения максимума из двух рациональных чисел;
   (e) функции *check*, которая проверяет, является ли заданное рациональное число несократимой дробью.

Указания: используйте явный стиль спецификации.

7. Определить функцию *approx_sqrt*, которая с заданной точностью *eps* (положительное вещественное число) находит приближённое значение корня квадратного из неотрицательного вещественного числа. Возвращаемый

функцией результат должен быть таким, чтобы точное значение квадратного корня лежало в полуоткрытом интервале:

[approx_sqrt(x,eps), approx_sqrt(x,eps) + eps).

Указания:

- здесь следует использовать неявный стиль описания, т.к. в постановке задачи оговариваются только соотношения между входными и выходными параметрами и не уточняется алгоритм вычисления приближенного значения квадратного корня;
- обратите внимание на то, что функция определена не для всех значений входных параметров, т.е. является частично вычислимой.

8. Дана явная спецификация функции, предложить неявную спецификацию функции, эквивалентной данной:

(a)
```
value
F1 : Int >< Int >< Int -> Int >< Int
F1(a, b, c) is      if a < b then (a * c, b - a)
                    elsif a > b then (a – b, b * c)
                    else (a, b)
                    end
```

(b)
```
value
F2 : Nat >< Nat >< Nat -> Nat >< Nat
F2(a, b, c) is      (if a + b > c then c else a + b end,
                    a * b * (if c > 0 then c else 0 –c end))
```

ГЛАВА 3. ОСНОВНЫЕ АБСТРАКЦИИ ДАННЫХ В МОДЕЛЕ-ОРИЕНТИРОВАННЫХ СПЕЦИФИКАЦИЯХ. МНОЖЕСТВА

Понятие множества, конечные и бесконечные множества. Способы определения множеств. Операции над множествами в RSL. Использование абстракции множеств в моделе-ориентированных спецификациях. Упражнения.

Понятие множества

Понятие множества в RSL вполне совпадает с математическим понятием множества, т.е. под *множеством* понимается неупорядоченный набор различных значений одного и того же типа. Например:

{1,3,5}
{0.0, 1.5, 4.7}
{″Mary″,″John″,″Peter″}

Из определения множества следуют два основных свойства множества:

- все элементы множества различны между собой,
- на множестве не задано отношение порядка.

Таким образом, выражения {1,3,5}, {5,3,1} и {3,5,1,1} определяют одно и то же множество.

В RSL различаются конечные и бесконечные множества. Для описания типа множества в RSL предусмотрены типовые операции **-set** и **-infset**, с помощью которых строится составной тип T-**set** для описания конечных множеств из элементов типа T и T-**infset** – для бесконечных множеств из элементов того же типа. Так, значениями типа **Bool-set** являются следующие конечные множества:

{}
{true}
{false}
{true, **false}**,

причем сюда включается пустое множество {}.

Тип T-**infset** представляет как конечные, так и бесконечные множества, состоящие из элементов типа T. Следовательно, для любого типа T тип T-**set** является подтипом T-**infset**. Например, тип **Nat-infset** содержит все конечные (в том числе и пустое) и бесконечные множества натуральных чисел.

Способы определения множеств

Конечное множество может быть задано путем непосредственного перечисления его элементов (именно этот способ был использован в приведенных выше примерах), в этом случае выражение, определяющее значение множества, имеет вид $\{v_1,...,v_n\}$, где $n \geq 0$. Кроме того, для конечных множеств, образованных из целых чисел, возможно также использование диапазона для задания значения множества, причем левая граница диапазона не должна превышать его правую границу (в противном случае множество будет пусто). Например, выражение {3 **..** 7} задает множество {3,4,5,6,7}, {3 **..** 3} – множество из единственного элемента {3} и {3 **..** 2} – пустое множество {}.

Более общей формой выражения, определяющего значение как конечного, так и бесконечного множества, является выражение вида:

```
{ value_expr | set_limitation },
```

называемое *сокращенной* (compehended) записью множества или сокращенным выражением. При этом выражение value_expr задает формулу для вычисления значений элементов множества, конструкция set_limitation определяет тип элементов множества и возможные ограничения на значения этих элементов, заданные в виде некоторого предиката. Примером такой формы записи может служить выражение {2∗n | n **: Nat :-** n **<=** 3}, с помощью которого задается конечное множество {0, 2, 4, 6} или выражение {2∗n | n **: Nat**}, определяющее бесконечное множество из четных натуральных чисел.

Операции над множествами

Для работы с множествами в RSL определены следующие операции:

inter – пересечение (∩),
union – объединение (∪),
\ - вычитание,
isin, **~isin** – проверка принадлежности (или не принадлежности) множеству (∈ ,∉),
<<, **<<=**, **>>**, **>>= -** проверка, является ли одно множество подмножеством другого (⊂, ⊆, ⊃, ⊇),
card – вычисление количества элементов множества,

кроме того можно использовать операции = и ~=.
Операция **card** вычисляет размер конечного множества, т.е. количество его элементов:

```
card : T-infset -~-> Nat
```

При применении к бесконечному множеству **card** возвращает **chaos**. Например:

```
card {1, 4, 56} = 3
card {} = 0
card {n | n : Nat} ≡ chaos
```

Упражнения

1. Определить множество, элементами которого являются:

   (a) нечетные числа в диапазоне от 0 до 10,
   (b) простые числа в диапазоне от 2 до 20.

Указания: используйте два способа задания значения множества - непосредственное перечисление его элементов и сокращенное выражение.

2. Вычислить значение выражения:
   card { n | n **: Nat** :- 30 \ n = 0}

3. Написать RSL выражение для вычисления количества точек с целочисленными координатами, попадающими внутрь круга с центром в начале координат и радиусом 5.

Указания: определите множество координат соответствующих точек и найдите количество его элементов.

4. Определить функцию:

   (a) *max*, возвращающую максимум из значений элементов непустого множества из целых чисел,
   (b) *choose*, возвращающую некоторый элемент непустого множества из элементов типа T.

Указания:

- используйте неявный стиль описания функций,
- обратите внимание на то, что обе функции определены не для всех значений входного параметра, т.е. являются частично вычислимыми.

5. Пример использования абстракции множеств в моделе-ориентированной спецификации. В базе данных университета содержится информация о студентах, обучающихся в этом университете, и читаемых им курсах. Пусть база данных университета описана следующим образом:

```
scheme
  UNIVERSITY_SYSTEM =
    class
      type
          Student,
          Course,
          CourseInfo = Course >< Student-set,
          University = {| (ss, cis) : Student-set >< CourseInfo-set :- is_wf(ss, cis) |}
      value
          /∗ is_wf осуществляет проверку на «хорошо сформированную» базу данных,
              проверяя выполнение свойства уникальности курса ∗/
          is_wf :- Student-set >< CourseInfo-set -> Bool,
          /* hasStudent проверяет, учится ли данный студент в указанном университете
              */
          hasStudent : Student >< University -> Bool,
          /∗ hasCourse проверяет, читается ли данный курс в указанном университете */
          hasCourse : Course >< University -> Bool,
          /∗ studOf возвращает множество студентов заданного университета,
              посещающих указанный курс ∗/
          studOf : Course >< University -~-> Student-set,
          /∗ attending возвращает множество курсов, которые посещает данный студент
              в указанном университете ∗/
          attending : Student >< University -~-> Course-set,
          /∗ newStud добавляет нового студента к числу студентов заданного
              университета ∗/
          newStud : Student >< University -~-> University,
```

```
        /* dropStud исключает указанного студента из числа студентов заданного
           университета */
        dropStud : Student >< University -~-> University,
        /* newCourse добавляет новый курс с пустым множеством студентов к числу
           курсов заданного университета */
        newCourse : Course >< University -~-> University,
        /* delCourse удаляет указанный курс из числа курсов заданного университета
           */
        delCourse : Course >< University -~-> University
    end
```

(a) Написать явное определение функций, сигнатура которых приведена в данном описании.
(b) Описать неявную спецификацию функции *studOf*.
(c) Почему некоторые из перечисленных выше функций частично вычислимы?

Указания:

- обратите внимание на необходимость введения ограничения подтипа при определении типа *University* для проверки того свойства, что для любого читаемого в университете курса в базе данных должно быть зафиксировано единственное множество посещающих этот курс студентов (свойство уникальности курса),
- в пункте (а) воспользуйтесь явным стилем описания функций и операциями над множествами.

ГЛАВА 4. ОСНОВНЫЕ АБСТРАКЦИИ ДАННЫХ В МОДЕЛЕ-ОРИЕНТИРОВАННЫХ СПЕЦИФИКАЦИЯХ. СПИСКИ

Понятие списка, конечные и бесконечные списки. Способы определения списков. Операции над списками. Выражение **case**. Использование абстракции списков в моделе-ориентированнных спецификациях. Упражнения

Понятие списка

Под *списком* понимается последовательность значений одного и того же типа, например:

<.1, 3, 3, 1, 5**.>**
<.true, **false**, **true.>**
<.3.14, 0.15**.>**

Отсюда следуют два основных свойства списка:

- порядок следования элементов в списке определен и существенен,
- допускается повторное вхождение элементов в список.

Таким образом, выражения **<.**1, 3, 5**.>**, **<.**5, 3, 1**.>** и **<.**1, 3, 3, 1, 5**.>** определяют три разных списка.

В RSL различаются конечные и бесконечные списки. Для описания типа списка предусмотрены типовые операции -**list** и -**inflist**, с помощью которых строится составной тип T-**list** для описания конечных списков из элементов типа T и T-**inflist** – для бесконечных списков из элементов того же типа. Например, значениями типа **Bool-list** являются все конечные списки (в том числе и пустой) из булевских значений.

Выражение T-**inflist** задает тип как конечных, так и бесконечных списков из элементов типа T. Следовательно, для любого типа T тип T- **list** является подтипом T-**inflist**.

Способы определения списков

Для списков применяются те же способы определения значений, что и для множеств. Так, значение конечного списка может быть задано путем непосредственного перечисления его элементов. В этом случае значение списка определяется выражением вида **<.**v_1,...,v_n**.>**, где $n \geq 0$ и все v_i являются выражениями одного и того же типа, в частности, **<..>** задает пустой список. В приведенных выше примерах был использован именно этот способ. Конечный список из последовательных целых чисел можно задать, указав диапазон изменения значений элементов списка, т.е. выражением вида **<.**v_1**..**v_2**.>**, где v_1 и v_2 задают соответственно нижнюю и верхнюю границы диапазона, причем при $v_1 > v_2$ список пуст.

Использование такого способа записи иллюстрируют следующие примеры:

<.3**..**7**.>** =**<.**3, 4, 5, 6, 7**.>**
<.3**..**3**.>** = **<.**3**.>**
<.3**..**2**.>** = **<..>**

Значение списка можно задать также по аналогии с множествами и с помощью так называемого сокращенного выражения (comprehended list expression), имеющего вид **<.** value_expr | list_limitation **.>**. Этот способ применяется в том случае, когда новый список строится на основе какого-то уже существующего. Здесь value_expr определяет общую формулу для вычисления значений элементов нового списка, list_limitation задает базовый список, на основе которого строится данный, с возможным указанием некоторого предиката для отбора элементов из базового списка.
Например, в выражении:

<.2∗n | n **in <.**0 **..** 3**.>.>**

базовым является список **<.**0 **..** 3**.>**, предикат отбора отсутствует и, следовательно, в результате вычисления получается список **<.**0,2,4,6**.>**, причем упорядоченность элементов нового списка полностью определяется порядком следования элементов в базовом списке. Примером использования предиката для отбора элементов базового списка может служить выражение:

<.n | n **in <.**1 **..** 100**.> :-** is_a_prime(n)**.>**,

где предикат is_a_prime(n) позволяет определить, является ли n простым числом. С помощью данного выражения задается список, элементами которого являются все простые числа из диапазона 1 **..** 100 в возрастающем порядке, т.е. **<.**2,3,5,7,...,97**.>**.

Для доступа к какому-либо конкретному элементу списка в RSL предусмотрено понятие индекса. В качестве индекса используются натуральные числа, причем индексация элементов списка начинается с 1 и для конечных списков заканчивается числом, равным длине списка. Например:

<.2,5,3**.>** (2) = 5
<.<.2,5,3**.>**, **<.**3**.>.>** (1) = **<.**2,5,3**.>**
<.<.2,5,3**.>**, **<.**3**.> .>** (1)(2) = **<.**2,5,3**.>** (2) = 5

В случае отсутствия в списке элемента с указанным индексом соответствующее выражение принимает значение **chaos**. Например:

<.1 **..** 50**.>** (51) = **chaos**

Значение бесконечного списка может быть задано с помощью аксиом, определяющих правила формирования списка. Так, список, содержащий все натуральные числа в возрастающем порядке, может быть определен в виде константы all_natural_numbers следующим образом:

```
value
  all_natural_numbers : Nat-inflist
axiom
  all_natural_numbers(1) = 0,
  exists idx : Nat :-
          idx >= 2 =>
            all_natural_numbers(idx) = all_natural_numbers(idx –1) + 1
```

На основе уже определенного бесконечного списка с помощью сокращенного выражения можно задавать новые бесконечные списки. Например, список,

элементами которого являются все простые числа в возрастающем порядке, может быть определен посредством выражения:

```
<.n | n in all_natural_numbers :- is_a_prime(n).>
```

Операции над списками

Над списками в RSL определены следующие операции:

```
^ : T-list >< T-inflist -> T-inflist
hd : T-inflist -~-> T
tl : T-inflist -~-> T-inflist
len : T-inflist -~-> Nat
elems : T-inflist -> T-infset
inds : T-inflist -> Nat-infset
```

Кроме того к спискам, как и к любому типу в RSL, применимы операции = и ~=.

Операция ^ означает конкатенацию двух списков, первый из которых обязательно должен быть конечным.

Результатом применения операций **hd** и **tl** являются соответственно головной элемент списка и «хвост» списка - оставшаяся после удаления головного элемента часть списка, причем обе эти операции определены только для непустого списка. Для любого непустого списка L верно соотношение:

```
hd L is L(1)
```

Операция **len** возвращает длину конечного списка (для пустого списка возвращается 0), при применении к бесконечному списку результатом является **chaos**.

Операция **elems** выдает в качестве результата множество, состоящее из элементов заданного списка. Например:

```
elems <..> = {}
elems <.1,3,1.> = {1, 3}
elems all_natural_numbers = {n | n : Nat}
```

Количество различных элементов списка L можно определить с помощью выражения **card elems** L.

Результатом применения операции **inds** является множество индексов заданного списка. Например:

```
inds <..> = {}
inds <.1,3,1.> = {1, 2, 3}
```

Таким образом, для конечного списка FL:

```
inds FL = {1 .. len FL},
```

для бесконечного списка IL:

```
inds IL = {idx | idx : Nat :- idx >= 1}.
```

Для произвольного списка L верно соотношение:

elems L **is** {L(x) | x **: Nat :-** x **isin inds** L}

Встроенный тип **Text** представляет собой конечный список из элементов типа **Char**, т.е. **Text = Char-list**, следовательно, к значениям этого типа применимы все операции со списками. Например:

“abc”(2) = **<.**’a’, ‘b’, ‘c’**.>**(2) = ‘b’
tl “abc” = “bc”

Выражение case

Достаточно часто при работе со списками используется выражение **case**, с помощью которого осуществляется выбор одного из нескольких различных вариантов значения некоторого выражения. В общем случае выражение **case** имеет следующий вид:

case value_expr **of**
  $pattern_1$ -> $value\_expr_1$,
    ...
  $pattern_n$ -> $value\_expr_n$
end, где $n \geq 1$.

Здесь выражение value_expr задает селектор выбора, $pattern_1$,…,$pattern_n$ – шаблоны подстановки, каждый из которых определяет соответствующий вариант значения выражения. При вычислении выражения значение селектора value_expr последовательно (сверху вниз) сопоставляется с указанными шаблонами до первого совпадения. В случае успешного сопоставления с шаблоном $pattern_i$ дальнейшее сопоставление не производится и в качестве результирующего значения всего выражения принимается значения выражения $value\_expr_i$. Если попытка сопоставления для всех шаблонов закончилась неуспешно, имеет место неполная спецификация (недоспецификация).

В RSL определено несколько видов шаблонов, мы ограничимся рассмотрением только трех из них. Простейшими видами шаблонов являются *литеральный шаблон* (в качестве шаблонов используются литералы RSL) и *шаблон универсальной подстановки* (_), сопоставление с которым всегда приводит к успешному результату. В силу своей специфики шаблон универсальной подстановки всегда должен располагаться последним в списке шаблонов выражения **case**. Использование указанных видов шаблонов иллюстрирует пример спецификации функции, вычисляющей числа Фибоначчи.

```
type Nat1 = {|n : Nat :- n > 0|}
value Fib: Nat1 -> Nat1
        Fib(n) is
            case n of
                    1 -> 1,
                    2 -> 1,
                    _ -> Fib(n - 2) + Fib(n - 1)
            end
```

Списочный шаблон представляет собой некоторое выражение списочного типа, основное назначение которого – ввести именование отдельных компонент данного выражения, т.е. шаблон фактически заменяет конструкцию **if** (**let ...**). Примером использования данного вида шаблонов может служить следующая спецификация функции, переставляющей элементы конечного целочисленного списка в обратном порядке:

```
value reverse : Int-list -> Int-list
        reverse(L) is
            case L of
                <..> -> <..>,
                <.i.> ^ Lr-> reverse(Lr) ^ <.i.>
            end
```

Упражнения

1. Вычислить значение выражения:

   (a) **<.**n * n | n **in <.**1..20**.> :-** n \ 2 = 1**.>**(7),
   (b) **<.**n * n | n **in <.**1..20**.> :-** n \ 2 = 1**.>**(12).

2. Определить список, элементами которого являются:

   (a) четные числа в диапазоне от 2 до 16,
   (b) все полные квадраты в диапазоне от 1 до 100.

Указания: используйте оба способа задания значения списка - непосредственное перечисление его элементов и сокращенное выражение.

3. Определить список, элементами которого являются числа Фибоначчи, не превосходящие 1000.

Указания:

- определите с помощью аксиом список из всех чисел Фибоначчи,
- используя построенный список в качестве базового, определите искомый список.

4. Определить функцию *is_sorted*, проверяющую конечный целочисленный список на упорядоченность по возрастанию.

Указания: воспользуйтесь аксиоматическим описанием функции.

5. Пусть конечный список образован из элементов типа *Elem*. Определить следующие функции:

   (a) *length* – вычисляет длину списка без использования встроенной операции **len**,
   (b) *rev* – переставляет элементы списка в обратном порядке,
   (c) *del* – удаляет из списка все вхождения заданного элемента,
   (d) *number_of* – подсчитывает количество вхождений в список заданного элемента.

Указания:

- используйте рекурсивное определение функций с помощью операций **hd** и **tl**;
- в рекурсивном определении воспользуйтесь условным выражением или выражением **case**,
- в пунктах (c) и (d) помимо рекурсивного определения функции предложите также выражение, с помощью которого вычисляется результат функции.

6. Определить функцию *max* для вычисления максимального элемента непустого конечного списка из вещественных элементов.

7. Предложить спецификацию следующей упрощенной модели системы обработки текстов: текст разделен на страницы, каждая страница состоит из строк каких-то слов. Определить схему *PAGE*, обеспечивающую описание:

   (a) типов *Page*, *Line*, и *Word* для описания соответственно страницы, строки и слова текста,
   (b) функции *is_on*, проверяющей, встречается ли указанное слово на заданной странице,
   (c) функции *number_of*, подсчитывающей количество вхождений указанного слова в текст заданной страницы.

Указания:
- в пункте (a) воспользуйтесь конечными списками для описания типов *Page* и *Line*,
- в пункте (b) используйте квантифицированное выражение и операции **inds** и **elems**,
- в пункте (c) опишите с помощью сокращенного выражения множество пар индексов (номер строки, номер слова в строке), определяющее все вхождения указанного слова в строки данной страницы, и воспользуйтесь операцией **card**.

ГЛАВА 5. ОСНОВНЫЕ АБСТРАКЦИИ ДАННЫХ В МОДЕЛЕ-ОРИЕНТИРОВАННЫХ СПЕЦИФИКАЦИЯХ. ОТОБРАЖЕНИЯ

Понятие отображения, конечные детерминированные отображения. Способы определения отображений. Операции над отображениями. Использование абстракции отображений в моделе-ориентированнных спецификациях. Упражнения.

Понятие отображения

Отображением называется неупорядоченный набор пар значений, который отображает значения одного типа в значения другого типа, при этом множество отображаемых значений называется *областью определения* или *доменом* (domain) отображения, а множество значений, в которые осуществляется отображение, называется его *областью значений* (range). Например:

[3 +> **true**, 5 +> **false**]
[″Klaus″ +> 7, ″John″ +> 2, ″Mary″ +> 7]

Первый пример задает отображение из натуральных чисел в значения типа **Bool**, второй – из типа **Text** в тип **Nat**. *Областью определения* (*доменом*) для первого отображения является множество {3,5}, *областью его значений* – множество {**true**,**false**}, для второго отображения такими множествами являются {″Klaus″,″John″,″Mary″} и {2,7}.

Каждая пара отображения задается выражением v +> w и определяет отображение ключа v в значение w. Порядок следования пар в отображении несущественен, поэтому, например, выражения [3 +> **true**,5 +> **false**] и [5 +> **false**,3 +> **true**] определяют одно и то же отображение.

Тип отображения задается типовым выражением:

T_1 –m-> T_2,

где T_1 - тип элементов домена отображения (ключей) и T_2 – тип элементов области значений отображения. Так, приведенные выше отображения имеют типы **Nat** -m->**Bool** и **Text** -m-> **Nat** соответственно.

Отображение может быть конечным или бесконечным в зависимости от того, конечным или бесконечным является набор пар, определяющий это отображение. Конечное отображение имеет вид:

[v_1 +> w_1,…,v_n +> w_n],

бесконечное отображение имеет вид:

[v_1 +> w_1,…,v_n +> w_n,…],

где $n \geq 0$, v_i и w_i – некоторые выражения типа T_1 и T_2 соответственно.

Отображение может быть детерминированным или недетерминированным при применении к элементам из его домена. Для детерминированного отображения справедливо свойство уникальности ключа, т.е. верно соотношение:

$v_i = v_k \Rightarrow w_i = w_k$,

где v_i и v_k – любые элементы из домена этого отображения.

Примером конечного детерминированного отображения может служить любое из приведенных выше отображений. В качестве примера конечного недетерминированного отображения можно привести отображение:

[3 +> **true**,3 +> **false**].

Способы определения отображений

Для определения отображений в RSL используются те же способы, что и для множеств и списков, а именно: непосредственное перечисление элементов и сокращенное выражение. Первый способ может применяться для конечных отображений, в этом случае отображение задается выражением вида:

$[v_1 +> w_1,\ldots,v_n +> w_n]$,

где $n \geq 0$ (в частности, при $n = 0$ имеем пустое отображение []). Здесь выражения v_i задают значения ключей и w_i – сопоставляемых им значений. Именно этот способ был использован во всех рассмотренных ранее примерах.

Сокращенное выражение (comprehended map expression) используется для определения значений как конечных, так и бесконечных отображений и имеет вид:

[value_expr_pair | set_limitation],

где выражение value_expr_pair задает общую формулу для определения входящих в отображение пар, set_limitation задает возможные ограничения на домен отображения. Например, значением сокращенного выражения:

[n +> 2∗n | n **: Nat :-** n <= 2]

является конечное отображение [0 +> 0,1 +> 2,2 +> 4].
Выражение:

[n +> 2∗n | n **: Nat**]

определяет бесконечное отображение [0 +> 0,1 +> 2,2 +> 4,…].

Значение отображения для какого-либо конкретного элемента из его домена (применение отображения) задается с помощью выражения $value\_expr_1(value\_expr_2)$, где $value\_expr_1$ является выражением, определяющим данное отображение, $value\_expr_2$ – выражение для вычисления значения некоторого элемента из домена отображения. Например:

[″Klaus″ +> 7, ″John″ +> 2, ″Mary″ +> 7](″John″) = 2,
[1 +> [″Per″ +> 5, ″Jan″ +> 7], 2 +> [ ]](1)(″Jan″) =
= [″Per″ +> 5, ″Jan″ +> 7] (″Jan″) = 7,

или для недетерминированного отображения:

[3 +> **true**,3 +> **false**](3) = **true** |^| **false,**

где символ |^| означает недетерминированный (внутренний) выбор из двух указанных значений. Для значений, не принадлежащих домену отображения, эффект применения отображения не определен, т.е. соответствующее выражение возвращает **chaos**:

[1 +> 2, 2 +> 3](5) = **chaos**

Операции над отображениями

Над отображениями определены следующие операции:

dom : (T_1 –m-> T_2) -> T_1**-infset**
rng : (T_1 –m-> T_2) -> T_2**-infset**
!! : (T_1 –m-> T_2) >< (T_1 –m-> T_2) -> (T_1 –m-> T_2)
union : (T_1 –m-> T_2) >< (T_1 –m-> T_2) -> (T_1 –m-> T_2)
\ : (T_1 –m-> T_2) >< T_1**-infset** -> (T_1 –m-> T_2)
/ : (T_1 –m-> T_2) >< T_1**-infset** -> (T_1 –m-> T_2)
 # : (T_2 –m-> T_3) >< (T_1 –m-> T_2) -> (T_1 –m-> T_3)

Кроме того для отображений определены операции = и ~=.

Операции **dom** и **rng** возвращают в качестве результата соответственно домен и область значений отображения, причем в зависимости от вида отображения эти множества могут быть конечными или бесконечными. Для любого отображения m верно соотношение:

rng m **is** {m(x) | x **:** T_1 **:-** x **isin dom** m}

Примеры:

dom [3 +> **true**, 5 +> **false**] = {3, 5}
rng [3 +> **true**, 5 +> **false**] = {**true**, **false**}
dom [ ] = { }
rng [ ] = { }
dom [n +> 2∗n | n **: Nat**] = { n | n **: Nat** }
rng [n +> 2∗n | n **: Nat**] = { 2*n | n **: Nat** }

Результатом операции !! (переопределение отображения) является набор пар, полученный объединением наборов пар первого и второго отображений, причем в случае пересечения доменов предпочтение отдается парам из второго отображения, т.е. для элементов, попавших в пересечение доменов отображений, второе отображение переопределяет первое. Определение этой операции имеет вид:

x !! y **is** [v +> w | v **:** T_1, w **:** T_2 **:-** w = y(v) \/ v **~isin dom** y /\ w = x(v)]

Эффект выполнения данной операции иллюстрируют следующие примеры:

[3 +> **true**, 5 +> **false**] !! [5 +> **true**] = [3 +> **true**, 5 +> **true**]
[3 +> **true**] !! [5 +> **false**] = [3 +> **true**, 5 +> **false**]
[3 +> **true**] !! [ ] = [3 +> **true**]

Операция **union** просто объединяет наборы пар двух заданных отображений, например:

[3 +> **true**, 5 +> **false**] **union** [5 +> **true**] = [3 +> **true**, 5 +> **false**, 5 +> **true**]

Эта операция может привести к недетерминизму (как в рассмотренном выше примере), поэтому во избежание возникновения недетерминизма обычно применяется к отображениям с непересекающимися доменами.

Операции ограничения отображений \ и / позволяют изменять домены отображений, а именно: \ удаляет из домена отображения указанное множество, / , напротив, оставляет в домене только те элементы, которые входят в указанное множество. Более точно определение этих операций выглядит следующим образом:

m \ s = [v +> m(v) | v **:** T_1 **:-** v **isin** (**dom** m) \ s]
m / s = [v +> m(v) | v **:** T_1 **:-** v **isin** (**dom** m) **inter** s]

Некоторые примеры:

[3 +> **true**, 5 +> **false**] \ {5, 7} = [3 +> **true**]
[3 +> **true**, 5 +> **false**] / {5, 7} = [5 +> **false**]
[3 +> **true**, 5 +> **false**] \ {3, 5, 7} = []
[3 +> **true**, 5 +> **false**] / {3, 5, 7} = [3 +> **true**, 5 +> **false**]

Операция # позволяет осуществлять композицию двух отображений, т.е. для отображений m_1 и m_2 она определяется так:

m_1 # m_2 = [v +> $m_1(m_2(v))$ | v **:** T_1 **:-** v **isin dom** m_2 /\ $m_2(v)$ **isin dom** m_1]

Например:

[3 +> **true**, 5 +> **false**] # [″Klaus″ +> 3, ″John″ +> 7] = [″Klaus″ +> **true**]
[3 +> **true**] # [″Klaus″ +> 5] = [ ]

Упражнения

1. Определить отображение, сопоставляющее:
   (a) каждому нечетному натуральному числу, не превосходящему 30, его остаток от деления на 3,
   (b) каждой паре натуральных чисел остаток от деления первого числа на второе (первое число должно быть меньше 50),
   (c) каждому натуральному числу, не превосходящему 20, все его делители,
   (d) каждому натуральному числу ближайшее, не превосходящее его, число, являющееся полным квадратом,
   (e) каждому натуральному числу n, не превосходящему 30, все простые числа из диапазона [2, n]; применить построенное отображение к n = 1, 10, 50.

Указания: используйте сокращенное выражение для определения отображений.

2. Пример использования абстракции отображений в моделе-ориентированной спецификации. Пусть база данных университета описана следующим образом:

```
scheme
  MAP_UNIVERSITY_SYSTEM =
    class
      type
        Student,
        Course,
        CourseInfos = Course -m-> Student-set,
        University = Student-set >< CourseInfos
      value
          /* hasStudent проверяет, учится ли данный студент в указанном университете
             */
          hasStudent : Student >< University -> Bool,
          /* hasCourse проверяет, читается ли данный курс в указанном университете
             */
          hasCourse : Course >< University -> Bool,
          /* studOf возвращает множество студентов заданного университета,
             посещающих указанный курс */
          studOf : Course >< University -~-> Student-set,
          /* attending возвращает множество курсов, которые посещает данный студент
             в указанном университете */
          attending : Student >< University -~-> Course-set,
          /* newStud добавляет нового студента к числу студентов заданного
             университета */
          newStud : Student >< University -~-> University,
          /* dropStud исключает указанного студента из числа студентов заданного
             университета */
          dropStud : Student >< University -~-> University,
          /* newCourse добавляет новый курс с пустым множеством студентов к числу
             курсов заданного университета */
          newCourse : Course >< University -~-> University,
          /* delCourse удаляет указанный курс из числа курсов заданного университета
             */
          delCourse : Course >< University -~-> University
    end
```

Определите функции, сигнатура которых приведена в данном описании.

Указания:

- воспользуйтесь явным стилем описания функций и операциями над отображениями,
- обратите внимание на тот факт, что использование в данной спецификации детерминированного отображения для типа *CourseInfos* автоматически обеспечивает свойство уникальности курса в предложенной моделе-ориентированной спецификации и, следовательно, отпадает необходимость введения ограничения подтипа.

ГЛАВА 6. АЛГЕБРАИЧЕСКИЕ СПЕЦИФИКАЦИИ

Понятие алгебраической спецификации. Классификация функций модели на генераторы, модификаторы и обсерверы. Методика построения алгебраической спецификации. RAISE метод разработки программ. Понятие уточнения моделей. Проверка согласованности моделей. Упражнения.

Понятие алгебраической спецификации

Алгебраическая спецификация представляет собой частный случай аксиоматической спецификации, когда с помощью аксиом описываются свойства не какой-либо отдельной функции, а некоторой цепочки определяемых функций. В общем случае такие аксиомы имеют вид:

id(value_expr$_1$, ..., value_expr$_n$) **is** value_expr,

где выражения value_expr$_i$ сами, как правило, содержат вызовы функций. Например, f(g(x),x) **is** h(g(x)).

При разработке спецификаций программной системы алгебраические спецификации используются на самом верхнем уровне абстракции, т.к., описывая свойства цепочек функций в терминах самих определяемых функций, не требуют уточнения типов и ограничиваются использованием только абстрактных типов.

Построение алгебраической спецификации системы начинается со спецификации сигнатур, где объявляются типы данных (используются абстрактные типы без конкретизации внутренней структуры) и сигнатуры функций. При этом среди объявленных типов особо выделяется тип, с помощью которого моделируется реализация целевой системы, - *целевой* тип.

В качестве примера рассмотрим разработку алгебраической спецификации простейшей СУБД со следующей функциональностью. База данных представляет собой набор записей, имеющих ключ и значение, причем каждое значение ключа должно встречаться в базе не более одного раза (свойство уникальности ключа). Система должна обеспечивать возможность добавления записи, удаления записи по ключу, проверки наличия в базе записи с заданным значением ключа, а также поиска значения по ключу, если запись с таким ключом имеется в базе.

Можно предложить следующую спецификацию сигнатур для данной системы:

```
DATABASE =
class
    type Database, Key, Data
    value
        /∗ пустая база данных∗/
        empty : Database,
        /∗ добавление записи в базу данных∗/
        insert : Key><Data><Database-> Database,
        /∗ удаление записи с указанным ключом из базы данных∗/
        remove : Key >< Database -> Database,
        /∗ проверка наличия в базе данных записи с заданным ключом ∗/
        defined : Key >< Database -> Bool,
        /∗ поиск значения по ключу, если запись с таким ключом в базе есть∗/
        lookup : Key >< Database -~-> Data
end
```

Целевым типом предложенной спецификации является тип Database.

Классификация функций модели на генераторы, модификаторы и обсерверы

На следующем шаге построения алгебраической спецификации осуществляется разбиение функций на два класса – генераторов (конструкторов) и обсерверов – в зависимости от их отношения к целевому типу. К классу *генераторов* (*конструкторов*) относятся функции, формирующие значения целевого типа. Формальным признаком таких функций является наличие целевого типа среди выходных параметров в сигнатуре функций. В рассмотренном примере этот класс функций представлен функциями empty, insert и remove. К классу обсерверов (от английского термина observers) относятся функции доступа к значениям целевого типа, возвращающие по значению целевого типа значения других типов. Такие функции не изменяют значения целевого типа. Формальным признаком обсерверов является наличие целевого типа только среди входных параметров. В рассмотренном примере этот класс функций представлен функциями defined и lookup.

Иногда из множества генераторов выделяют минимальное подмножество функций, при помощи которых можно получить любое значение целевого типа, и только эти функции называют генераторами или конструкторами, а остальные – *модификаторами* или преобразователями. Для нашего случая генераторами в этом понимании являются функции empty и insert, модификатором - remove. Действительно, любая база данных может быть представлена выражением вида:

insert(k1,d1,insert(k2,d2,...,insert(kn,dn,empty)...)),

т.е. может быть построена путем конечного числа применений функции insert к пустой базе данных.

Наличие модификаторов в числе генераторов избыточно с математической точки зрения построения базы данных, однако это удобно и наглядно для пользователя.

Методика построения алгебраической спецификации

Завершающим этапом построения алгебраической спецификации является построение набора аксиом, описывающих свойства цепочек определяемых функций. Мы ограничимся рассмотрением цепочек из двух функций. На практике в большинстве случаев достаточно включить в набор аксиомы для пар вида:

- обсервер / генератор,
- обсервер / модификатор.

Для полноты описания свойств целевого типа в ряде случаев рекомендуется расширить этот набор аксиомами для пар вида модификатор / генератор.

При составлении набора аксиом необходимо принимать во внимание возможную частичную определенность функций. Так, из-за частичной определенности функции lookup из состава аксиом следует исключить пару lookup(empty) и указать предусловие ко всем аксиомам с функцией lookup.

Итак, общая схема построения алгебраической спецификации:

1. специфицировать сигнатуры функций,
2. выделить генераторы и обсерверы,
3. составить набор аксиом для пар вида обсервер/генератор и обсервер/модификатор.

Для рассмотренного примера получим следующую алгебраическую спецификацию:

```
scheme DATABASE =
  class
      type Database, Key, Data
      value
          empty : Database,
          insert : Key><Data><Database-> Database,
          remove : Key >< Database -> Database,
          defined : Key >< Database -> Bool,
          lookup : Key >< Database -~-> Data
      axiom
          [ defined_empty ]
          all k:Key :-
              defined(k,empty) is false,
          [ defined_insert ]
          all k,k1:Key, d:Data, db:Database :-
              defined(k,insert(k1,d,db)) is
                  k = k1 \/ defined(k,db),
          [ defined_remove ]
          all k,k1:Key, db:Database :-
              defined(k,remove(k1,db)) is
                  k ~= k1 /\ defined(k,db),
          [ lookup_insert ]
          all k,k1:Key, d:Data, db:Database :-
              lookup(k,insert(k1,d,db)) is
                  if k = k1 then d
                            else lookup(k,db) end
              pre k = k1 \/ defined(k,db),
          [ lookup_remove ]
          all k,k1:Key, db:Database :-
              lookup(k,remove(k1,db)) is
                  lookup(k,db)
              pre k ~= k1 /\ defined(k,db)
  end
```

RAISE метод разработки программ. Понятие уточнения моделей

Разработка программных систем методом RAISE базируется на парадигме последовательного уточнения. Общая идея заключается в следующем:

- Разработка разбивается на шаги
- Сначала описывается максимально простая и абстрактная модель
- На каждом шаге строится более подробная и конкретная модель
- На каждом шаге проверяется согласованность моделей

Таким образом, разработка идет сверху вниз по шагам, от более абстрактных моделей к более конкретным. Процесс перехода от одной абстрактной модели к

другой называется *уточнением* (refinement), полученная в результате уточнения модель называется *реализацией*. При этом говорят, что одна спецификация *уточняет* другую, если:

- реализация сохраняет объявления всех сущностей исходной спецификации (абстрактные типы могут заменяться описанием типа, подтипы могут заменяться своими максимальными типами),
- могут появляться объявления и описания новых сущностей и свойств,
- все свойства (аксиомы) исходной спецификации остаются справедливыми в реализации.

Каждое уточнение модели сопровождается обязательной проверкой согласованности этого уточнения, для чего используется ограниченный набор формальных правил.

RAISE метод предполагает следующую последовательность разработки спецификаций. На первом шаге строится алгебраическая спецификация системы, оперирующая абстрактными типами данных, далее выполняется уточнение используемых типов и построение раздельных спецификаций для функций системы. Таким образом, в зависимости от выбранного уточнения типов строится та или иная моделе-ориентированная спецификация системы, оперирующая уже конкретными типами данных. При этом, как правило, сначала строится неявная спецификация функций системы, затем явная. На заключительном этапе разработки возможна автоматическая кодогенерация в целевой язык программирования.

Проверка согласованности моделей

Для проверки согласованности уточнения необходимо убедиться, что все свойства (аксиомы) исходной модели остаются справедливыми в уточненной модели. Для этого в RAISE применяется техника re-writing формального доказательства, базирующаяся на следующих механизмах:

1. правила вывода - набор теорем для выполнения:
   - тождественных преобразований,
   - преобразований кванторов,
2. подстановки - раскрытие идентификаторов и т.д.,
3. вычисление выражений.

Доказательство согласованности двух спецификаций (доказательство отношения «уточняет») для каждого свойства исходной модели представляет собой цепочку вида:

$цель_1$
$[[аргументация_1]]$
…
$цель_n$
$[[аргументация_n]]$,

где очередная цель задает доказываемое свойство, аргументация задает ссылку на применяемое на данном шаге доказательства правило. Любая следующая цель в этой цепочке является результатом преобразования предыдущей цели в соответствии с указанным правилом.

В качестве примера рассмотрим доказательство справедливости аксиомы [defined_empty] для модели-реализации, где база данных моделируется в виде множества записей со следующим определением константы empty и функции defined:

```
empty : Database = { },
defined : Key >< Database -> Bool
defined(k,db) is (exists d : Data :- (k,d) isin db)
```

Пример доказательства для аксиомы [defined_empty].

```
[ defined_empty ]
all k:Key :-
        defined(k,empty) is false              - исходная цель доказательства
[[раскрыть квантор]]
      defined(k,empty) is false
[[раскрыть идентификатор empty]]
      defined(k,{}) is false
[[раскрыть идентификатор defined]]
      (exists d : Data :- (k,d) isin {}) is false
[[вычислить isin {}]]
      (exists d : Data :- false) is false
[[преобразовать квантор]]
      false is false
[[тождественное преобразование]]
      true
```

Упражнения

1. Построить алгебраическую спецификацию стека со следующими функциями:
   - поместить элемент в стек,
   - удалить верхний элемент непустого стека,
   - проверить стек на пустоту,
   - найти значение верхнего элемента непустого стека (без удаления из стека).

2. Построить алгебраическую спецификацию очереди со следующими функциями:
   - добавить элемент в очередь,
   - удалить элемент из непустой очереди,
   - проверить очередь на пустоту,
   - найти значение первого элемента непустой очереди (без удаления из очереди),
   - подсчитать количество элементов в очереди.

Указания: в упражнениях 1, 2 руководствуйтесь общей схемой построения алгебраической спецификации.

3. Используя абстракцию списков, уточнить построенную алгебраическую спецификацию стека в моделе-ориентированную спецификацию. Проверить согласованность уточнения.

4. Используя абстракцию списков, уточнить построенную алгебраическую спецификацию очереди в моделе-ориентированную спецификацию. Проверить согласованность уточнения.

5. Доказать, что вторая спецификация является (или не является) уточнением первой.

```
value
  f1 : A >< L -~-> L,
  f2 : A >< L -~->,
  f3 : L -> Nat
axiom
  all a : A, b : L :-
      f3(f1(a,b)) is f3(b) + 1
      pre f3(b) = 1,
  all a : A, b : L :-
      f3(f2(a,b)) is 1 + f3(b)
      pre f3(b) = 1,
  all a : A, b : L :-
      f1(a,b) is f2(a,b)
      pre f3(b) = 1
```

```
type
  L = A-list
value
  f1 : A >< L -~-> L
      f1(a,b) is <. a .> ^ tl b
      pre b ~= <..>,
  f2 : A >< L -~-> L
      f2(a,b) is tl b ^ <. a .>
      pre b ~= <..>,
  f3 : L -> Nat
      f3(ls) is len ls
```

ГЛАВА 7. ВАРИАНТНЫЕ ОПРЕДЕЛЕНИЯ

Понятие вариантного определения. Виды вариантных определений. Конструкторы, деструкторы и реконструкторы. Упражнения.

Понятие вариантного определения

Вариантное определение (определение вариантного типа) задается конструкцией вида:

type id = = variant$_1$ | ... | variant$_n$, n ≥ 1

и предназначено для удобного и компактного определения абстрактного типа (id) вместе с набором функций и констант над этим типом. Т.е. вариантное определение представляет собой сокращенную форму записи определения абстрактного типа, определения некоторых функций и аксиом. Среди определяемых функций могут быть:

- *конструкторы* – для генерации значений данного типа,
- *деструкторы* – для декомпозиции значений данного типа,
- *реконструкторы* – для модификации значений данного типа.

Аксиомы определяют свойства конструкторов, деструкторов и реконструкторов. Важной аксиомой среди них является аксиома индукции, которая гласит, что данный абстрактный тип полностью генерируется указанными конструкторами, т.е. любое значение этого типа получается в результате конечного числа применений конструкторов.

Виды вариантных определений. Конструкторы, деструкторы и реконструкторы

Конструкторы-константы

Простейшим видом вариантного определения является использование *конструкторов-констант*, когда тип определяется путем явного перечисления его значений. В качестве примера рассмотрим вариантное определение:

```
type Colour = = black | white
```

Оно представляет собой сокращенную форму записи следующего фрагмента спецификации:

```
type Colour                    /* объявление абстрактного типа */
value
    black : Colour,            /* определение констант */
    white : Colour
axiom
    [ disjoint ]
        black ~= white,
    [ induction ]
        all p : Colour -> Bool :- (p(black) /\ p(white)) => (all c : Colour :- p(c))
```

Первая аксиома гарантирует различие значений black и white. Вторая (аксиома индукции) позволяет доказать, что в типе Colour нет других значений кроме

указанных двух, т.е. данный тип полностью определяется указанными константами. С помощью аксиомы индукции можно индуктивно доказывать свойства над определяемым типом. В частности, для рассмотренного примера это означает, что если справедливость некоторого свойства доказана для значений black и white, то из этого следует справедливость этого свойства для всех значений типа Colour.

Конструкторы записей

Более сложным видом вариантного определения является использование в качестве конструкторов функций для генерации значений определяемого типа - *конструкторов записей* (record constructions). Такие конструкторы используются при определении структур данных. В частности, эти конструкторы могут использоваться для генерации записей, где под записью подразумевается набор именованных полей.

В качестве примера рассмотрим вариантное определение:

```
type Collection = = empty | add(Elem, Collection),
```

в котором используется конструктор-константа empty и конструктор записи add. Это определение рекурсивно определяет тип Collection, содержащий два вида значений:

1. значение empty,
2. значения, построенные с помощью конструктора add.

Данное вариантное определение представляет собой сокращение следующего фрагмента спецификации:

```
type Collection
value
    empty : Collection,
    add : Elem ><Collection -> Collection
axiom
    [ disjoint ]
        all el : Elem, c : Collection :- empty ~= add(el,c),
    [ induction ]
        all p : Collection -> Bool :-
            (p(empty) /\ (all el : Elem, c : Collection :- p(c) => p (add(el,c)))) =>
                (all c : Collection :- p(c))
```

Деструкторы

Деструкторами называются функции, предназначенные для выделения компонент из значений вариантного типа, с помощью таких функций осуществляется декомпозиция значений вариантного типа.

В качестве примера рассмотрим вариантное определение:

```
type List = = empty | add(head : Elem, tail : List),
```

где появляются два деструктора head и tail, обеспечивающие возможность декомпозиции значений, сгенерированных конструктором add, на «голову» и «хвост».

Данное вариантное определение является сокращением следующего фрагмента спецификации:

```
type List
value
    empty : List,
    add : Elem >< List -> List,
    head : List -~-> Elem,
    tail : List -~-> List
axiom
    [ disjoint ]
        all el : Elem, ls : List :- empty ~= add(el,ls),
    [ induction ]
        all p : List -> Bool :-
            (p(empty) /\ (all el : Elem, ls : List :- p(ls) => p (add(el,ls)))) =>
                (all ls : List :- p(ls))
    [ head_add ]
        all el : Elem, ls : List :- head(add(el,ls)) is el,
    [ tail_add ]
        all el : Elem, ls : List :- tail(add(el,ls)) is ls
```

Следует обратить внимание на частичность деструкторов head и tail, поскольку согласно исходному вариантному определению они определены только над значениями, сгенерированными конструктором add.

Реконструкторы

Реконструкторами называются функции, предназначенные для изменения значений отдельных компонент значений вариантного типа. С помощью реконструкторов осуществляется модификация значений вариантного типа.

В качестве примера рассмотрим предыдущее определение типа List с добавлением реконструктора replace_head для обеспечения возможности изменения головы списка.

```
type List = = empty | add(head : Elem <-> replace_head, tail : List)
```

Вхождение реконструктора добавит к вышеприведенной спецификации следующий фрагмент:

```
value
    replace_head : Elem >< List -~-> List
axiom
    [ head_ replace_head ]
        all el,el1 : Elem, ls : List :- head(replace_head(el,add(el1,ls))) is el,
    /* изменилось значение головы списка */
    [ tail_ replace_head ]
        all el : Elem, ls : List :- tail(replace_head(el,ls)) is tail(ls)
            pre ls ~= empty
    /* реконструктор replace_head не меняет хвост списка */
```

Первая аксиома выражает эффект изменения значения отдельной компоненты путем применения соответствующего реконструктора. Вторая аксиома выражает тот факт, что изменение реконструктором значения некоторой компоненты не оказывает никакого эффекта на другие компоненты.

Упражнения

1. Считая определенными абстрактные типы Key и Data, построить вариантное определение записи с полями «ключ» и «значение». Привести эквивалентное определение без использования вариантных определений.

Указания: используйте конструктор записей и деструкторы для декомпозиции записи на значения ее полей.

2. Считая определенным абстрактный тип Elem, построить вариантное определение стека. Привести эквивалентное определение без использования вариантных определений.

Указания: воспользуйтесь рекурсивным определением стека.

3. Для следующих вариантных определений привести эквивалентные определения без использования вариантных определений.

   (a) **type** Elem,
   Collection = = empty | add1(Elem, Collection) |
   add2(Elem, Elem, Collection)

   (b) **type** Elem,
   Tree = = empty | node(left : Tree, val : Elem, rigth : Tree)

   (c) **type** Elem,
   Tree = = empty | node(left : Tree, val : Elem <-> repl_val, rigth : Tree)

   (d) **type** Figure = = box(length **: Real**, width **: Real**) | circle(radius **: Real**) |
   triangle(base_line : **Real**, left_angle : **Real**, right_angle : **Real**)

ГЛАВА 8. ИМПЕРАТИВНЫЙ СТИЛЬ СПЕЦИФИКАЦИЙ

Понятие императивного стиля спецификаций. Описание переменных. Выражение присваивания. Функции с доступом к переменным. Императивные конструкции RSL. Описание локальных переменных. Явные и неявные спецификации в императивном стиле. Упражнения.

Понятие императивного стиля спецификаций

Спецификации в языке RSL могут разрабатываться как в аппликативном, так и в императивном стиле. *Аппликативный стиль* ближе к функциональному стилю программирования, именно этот стиль был использован во всех рассмотренных ранее спецификациях. *Императивный стиль* ближе к классическому понятию алгоритма как последовательности шагов его выполнения. Императивный стиль спецификации удобно использовать в том случае, когда данная спецификация будет реализовываться на императивном (не функциональном) языке программирования. В этом случае императивная спецификация выглядит более естественно, т.к. максимально приближена к языку реализации.

Характерной особенностью императивного стиля является использование *переменных* как способа передачи информации от одного шага выполнения алгоритма к другому. В RSL переменная рассматривается как контейнер, способный хранить значения определенного типа. Значение переменной может быть изменено путем присваивания ей нового значения. Совокупность значений всех объявленных в данной спецификации переменных формирует понятие *состояния* спецификации.

Описание переменных

Описание переменных в языке RSL производится в разделе **variable**, при этом каждое отдельное описание переменной имеет вид:

```
id : type_expr := value_expr,
```

где id задает имя переменной, type_expr – ее тип, выражение value_expr определяет начальное значение переменной. Инициализация переменной ее начальным значением является необязательной, и если она не указана, начальным значением переменной считается произвольное значение данного типа. Однотипные переменные без начального значения могут объявляться в одном описании. Например:

```
variable n : Nat := 1,
         s : Real = 0.0,
         a, b : Int
```

Областью видимости переменной считается весь модуль, в котором данная переменная описана.

Выражение присваивания

Для изменения значения переменной в RSL используется выражение присваивания вида:

id := value_expr

В качестве побочного эффекта при вычислении такого выражения переменной id присваивается значение выражения value_expr, при этом само выражение присваивания возвращает значение () типа **Unit**. Типы переменной id и выражения value_expr должны быть согласованы, причем тип выражения value_expr не может быть типом **Unit**. Пример выражения присваивания:

counter := counter +1

Функции с доступом к переменным

В сигнатуре функций, имеющих доступ к переменным, обязательно должно указываться описание доступа к этим переменным в форме:

write id_1, ..., id_n, $n \geq 1$,

если функция имеет доступ к перечисленным переменным на запись (и чтение), или в форме:

read id_1, ..., id_n, $n \geq 1$,

если указанные переменные доступны только на чтение. Если функция имеет доступ к переменным на запись, то в качестве своего побочного эффекта она может изменять значения данных переменных.

В качестве примера рассмотрим следующий фрагмент спецификации:

```
variable n, m : Nat
value
    f1 : Unit -> write n Nat,
    f2 : Unit -> read m Nat,
    f3 : Nat -> write n read m Nat
```

Здесь функции f1 и f2 не имеют входных параметров и, следовательно, зависят только от конкретного состояния, в котором происходит их вычисление. При этом f1 имеет доступ к переменной n на чтение и запись, f2 – к переменной m только на чтение. Функция f3 имеет доступ к переменной n на чтение и запись и к переменной m только на чтение.

Императивные конструкции RSL

Последовательность выражений

Последовательная композиция двух выражений задается выражением вида:

$value\_expr_1$; $value\_expr_2$

Вычисление такого выражения начинается с вычисления выражения $value\_expr_1$ с целью достижения его возможного побочного эффекта в виде изменения значений

входящих в него переменных. Затем уже в измененном состоянии вычисляется выражение value_expr$_2$. В качестве результата вычисления последовательной композиции выражений возвращается значение выражения value_expr$_2$. Выражение value_expr$_1$ должно иметь тип **Unit**. Например:

x := 1; x - выражение возвращает значение 1,
x := x + 1; x - выражение возвращает увеличенное на 1 значение переменной x.

В общем случае произвольная последовательность выражений задается выражением:

value_expr$_1$; ... ; value_expr$_n$, где n > 1.

При вычислении такого выражения последовательно слева направо вычисляются все входящие в него выражения, в качестве результата вычисления возвращается значение выражения value_expr$_n$. Все выражения кроме последнего должны иметь тип **Unit**.

Выражение if

Выражение **if** в императивных спецификациях имеет традиционную форму:

if value_expr **then** value_expr$_1$ **else** value_expr$_2$ **end**,

спецификой является только то, что выражения value_expr$_1$ и value_expr$_2$ имеют тип **Unit**. Кроме того возможна сокращенная форма выражения в виде:

if value_expr **then** value_expr$_1$ **end**

Эта форма эквивалентна выражению:

if value_expr **then** value_expr$_1$ **else skip end,**

где **skip** представляет предопределенное выражение типа **Unit**, не имеющее побочного эффекта.

Например:

if counter > 0 **then** counter := counter - 1 **end**

Конструкции циклов

В императивных спецификациях RSL используются три конструкции циклов для повторяющегося вычисления выражений, указанных в теле цикла:

- цикл с предусловием (**while**),
- цикл с постусловием (**until**),
- цикл с заранее известным числом повторений (**for**).

Выражение, задающее любую конструкцию цикла, имеет тип **Unit**, назначением такого выражения является повторяющееся вычисление указанных в теле цикла выражений с целью достижения их побочного эффекта, выражающегося в изменении значений переменных. Семантика конструкций цикла аналогична семантике соответствующих конструкций в языках программирования.

Цикл while

Конструкция цикла **while** представлена выражением **while** вида:

while value_expr **do** value_expr$_1$ **end**,

где выражение value_expr (тип **Bool**) задает условие повторения цикла, выражение value_expr$_1$ (тип **Unit**) – тело цикла (обычно последовательность выражений). Вычисление выражения **while** состоит из выполнения последовательных итераций. Каждая итерация цикла начинается с вычисления условия value_expr, и в случае его истинности выполняется вычисление выражения value_expr$_1$, после чего итерация повторяется, в противном случае вычисление выражения **while** завершается.

Цикл until

Конструкция цикла **until** представлена выражением **until** вида:

do value_expr$_1$ **until** value_expr **end**

Назначение выражений value_expr и value_expr$_1$ аналогично их назначению в предыдущей конструкции с той лишь разницей, что value_expr обозначает условие завершения цикла и в случае истинности этого условия вычисление выражения **until** завершается.

Цикл for

Конструкция цикла **for** представлена выражением **for** вида:

for list_limitation **do** value_expr$_1$ **end**,

где list_limitation задает список (возможно пустой) значений переменной цикла. Тело цикла (выражение value_expr$_1$) последовательно вычисляется для всех значений из указанного списка. Например:

```
for i in <. 1 .. n .> do
      result := result + 1.0 / (real i)
  end
```

Описание локальных переменных

В императивных спецификациях часто бывает удобно использовать локальные переменные. Объявление локальной переменной имеет вид:

local
  variable id : t := value_expr
in
  value_expr$_1$; ... ; value_expr$_n$
end, где $n \geq 1$.

Такая конструкция задает блок, внутри которого описывается локальная переменная id типа t с начальным значением, определяемым выражением value_expr (указывать начальное значение необязательно). Областью видимости локальной переменной является блок, в котором она описана.

Аналогичным образом можно локально описывать типы, функции и аксиомы.

Явные и неявные спецификации в императивном стиле

В качестве примера явной спецификации в императивном стиле приведем спецификацию функции increase, увеличивающей на 1 значение счетчика, представленного переменной counter.

```
variable counter : Nat := 0
value
    increase : Unit -> write counter Nat
    increase() is
        counter := counter + 1; counter
```

При составлении неявных спецификаций в императивном стиле возникает следующая проблема: если функция имеет побочный эффект, то при специфицировании результата ее выполнения необходимо уметь различать значения соответствующих переменных в состоянии до вызова функции и после него. Для этого в RSL предусмотрено выражение id', которое обозначает значение переменной id в состоянии до вызова функции.

Как пример такой спецификации рассмотрим неявную спецификацию функции choose, возвращающей в качестве результата произвольный элемент множества с одновременным удалением этого элемента из множества:

```
variable set1 : Int-set
value
    choose : Unit -~-> write set1 Int
    choose() as res
          post res isin set1' /\ set1 = set1' \ {res}
        pre set1 ~= { }
```

Первая часть постусловия (res **isin** set1') выражает тот факт, что до вызова функции элемент присутствовал в множестве, вторая (set1 = set1' \ {res}) гласит, что новое значение переменной set1 должно совпадать с ее значением до вызова функции за исключением удаляемого элемента.

Итак, при разработке неявных спецификаций в императивном стиле необходимо учитывать следующую рекомендацию. В спецификациях функций с побочным эффектом, имеющих доступ к переменным по записи, необходимо в постусловии специфицировать эффект изменения этих переменных. В частности, если изменения значения переменной не происходит, специфицировать этот факт выражением вида:

```
x = x'
```

Упражнения

1. Упростить выражения:

   (a) x := 1; x := 2
   (b) (x :=1; x) + (x := 2; x)
   (c) (x :=2; x) + (x := 1; x)

2. Построить моделе-ориентированную спецификацию стека в императивном стиле. Спецификация должна описывать следующие функции:

   - пустой стек,
   - поместить элемент в стек,
   - удалить верхний элемент непустого стека,
   - проверить стек на пустоту,
   - найти значение верхнего элемента непустого стека (без удаления из стека).

   (a) Использовать явное описание функций.
   (b) Использовать неявное описание функций.

Указания:

- для представления стека используйте переменную списочного типа,
- в сигнатуре функций опишите соответствующие права доступа к этой переменной.

3. Построить явную спецификацию в императивном стиле функции *fract_sum*, вычисляющей частичную сумму натурального ряда, используя:

   (a) конструкцию цикла **while**,
   (b) конструкцию цикла **until**,
   (c) конструкцию цикла **for**.

4. По явной спецификации функции построить ее неявную спецификацию:

   (a)
```
value
    f : Int >< Int >< Int -> write x, y Int >< Int
    f(a,b,c) is
        if a = b then
                (if a + b > c then c else x := y; a + b end,
                b * (if c > 0 then x :=c; c else 0 – c end))
            else (y :=a + b; y,a - b) end
```
   (b)
```
variable v1, v2 : Int
value
    f : Int >< Bool > -> write v1read v2 Nat >< Bool
    f(a,b) is
        local variable n : Int  in
            if b then v1 := v2 * a; n := v1 ** 2 + v2 ** 2
                else n := if v1 > v2 then v1 ** 2 – v2 ** 2
                                    else v2 ** 2 – v1 ** 2 end
            end;
            if v1 > v2 tnen v1 := v2 end;
            (n, v1 = v2)
        end
```

ГЛАВА 9. СПЕЦИФИКАЦИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

Каналы и взаимодействие процессов. Функции с доступом к каналам. Выражения взаимодействия. Композиция параллельных выражений: параллельная композиция, внутренний выбор, внешний выбор, взаимная блокировка. Упрощение параллельных выражений. Упражнения.

Каналы и взаимодействие процессов

RSL предоставляет возможность специфицировать параллельные процессы за счет имеющихся в языке средств для спецификации параллельного вычисления выражений. При этом вычисляющиеся параллельно выражения могут взаимодействовать между собой посредством *каналов*, что обеспечивается наличием в языке соответствующих примитивов взаимодействия.

Объявление каналов производится в разделе **channel**. Каждое отдельное описание этого раздела содержит перечисление имен каналов с указанием типа передаваемых посредством них значений и имеет вид:

$id_1, \ldots, id_n$: type_expr, где $n \geq 1$.

Например:

```
channel a : Nat,
        c, b : Int
```

Функции с доступом к каналам

Функции, имеющие доступ к каналам, в RSL называются также *процессами*. В сигнатуре таких функций доступ к каналам описывается в форме:

in $id_1, ..., id_n$, $n \geq 1$,

где перечисляются *каналы ввода*, из которых функция может получать значения, или в форме:

out $id_1, ..., id_n$, $n \geq 1$,

где указываются *каналы вывода*, по которым функция может выводить значения.
В качестве примера рассмотрим следующий фрагмент спецификации:

```
channel a ,b: Nat
value p : Unit -> in a out b Unit
```

Здесь функция p может вводить значения типа **Nat** по каналу a и выводить значения того же типа по каналу b.

Выражения взаимодействия

В RSL предусмотрено два примитива взаимодействия: один для ввода значения из канала, другой для вывода значения в канал. Если имеется следующее объявление канала:

```
channel id: T
```

то ввод значения из этого канала специфицируется *выражением ввода* вида:

```
id?
```

Вычисление данного выражения приостанавливается до тех пор, пока в указанный канал не будет выведено значение каким-либо другим процессом. В этом случае вычисление выражения ввода завершается, и принятое по каналу значение возвращается как результат вычисления выражения ввода. Таким образом, для завершения вычисления выражения ввода должно произойти взаимодействие процессов посредством указанного в выражении канала. Тип выражения ввода совпадает с типом канала.

Например, выражение:

```
a := c?
```

специфицирует факт присваивания переменной a значения, введенного по каналу c.

Вывод значения в канал специфицируется *выражением вывода* вида:

```
id! value_expr
```

Вычисление такого выражения начинается с вычисления выражения value_expr, значение которого выводится в канал id. На этом вычисление выражения вывода приостанавливается до тех пор, пока выведенное значение не будет потреблено из канала каким-либо другим процессом. Таким образом, для завершения вычисления выражения вывода также должно произойти взаимодействие процессов посредством указанного в выражении канала. Тип выражения value_expr должен совпадать с типом канала id. Типом выражения вывода является тип **Unit**.

В качестве примера рассмотрим спецификацию процесса opb, моделирующего одноместный буфер, взаимодействующий с окружением посредством каналов add и get. Значения типа Elem вводятся из канала add и затем выводятся в канал get.

```
scheme ONE_PLACE_BUFFER =
    class
        type Elem
        channel add, get : Elem
        value opb : Unit -> in add out get Unit
        axiom opb() is let v = add? in get!v end; opb()
    end
```

Данный процесс циклически выполняет указанное действие, цикличность обеспечивается рекурсивным вызовом процесса.

Композиция параллельных выражений

В RSL предусмотрено четыре вида композиции параллельных выражений:

- параллельная композиция,
- внешний выбор,
- внутренний выбор,
- взаимная блокировка (interlock).

Параллельная композиция

Параллельная композиция двух выражений задается выражением вида:

$value\_expr_1$ || $value\_expr_2$

Указанные выражения вычисляются параллельно до тех пор, пока одно из них не завершится, в то время как другое продолжит свое вычисление. Оба выражения должны иметь тип **Unit**, этот же тип имеет и всё выражение. Символ || обозначает комбинатор параллельной композиции.

Например, пусть имеются следующие объявления:

```
channel c : Int
variable x : Int
```

Тогда параллельная композиция выражений x := c? и c!5 задается выражением:

```
x := c? || c!5
```

Вычисление этого выражения может привести к потенциальному взаимодействию входящих в него выражений. В этом случае эффект вычисления параллельной композиции определяется выражением:

```
x := 5
```

Однако параллельные попытки записи в канал и чтения из канала необязательно приводят к взаимодействию. Это зависит от внутреннего выбора выражений. В частности, выражения могут вообще не осуществить никакого взаимодействия или осуществить взаимодействие с каким-либо другим параллельно вычисляющимся выражением. Так, вычисление выражения:

```
(x := c? || c!5) || c!7
```

может привести к одному из следующих эффектов:

- x := 7; c!5
- x := 5; c!7
- взаимодействие не происходит

Комбинатор параллельной композиции обладает свойствами коммутативности и ассоциативности.

Внешний выбор

Внешний выбор задается выражением вида:

$value\_expr_1$ |=| $value\_expr_2$

Выражения $value\_expr_1$ и $value\_expr_2$ должны иметь одинаковый тип, который считается также и типом всего выражения внешнего выбора. В результате внешнего выбора из двух указанных выражений для вычисления выбирается только одно, причем на выбор может оказать влияние окружение, т.е. другие выражения, вычисляющиеся параллельно с данным. Именно поэтому в названии данной композиции фигурирует термин «внешний». Символ |=| обозначает комбинатор внешнего выбора.

Внешний выбор служит для спецификации выбора между различными видами взаимодействия. Например, в области видимости объявлений:

channel c , d: **Int**
variable x : **Int**

рассмотрим выражение внешнего выбора:

x := c? |=| d!5

Это выражение предлагает два вида взаимодействия: ввод из канала c или вывод в канал d. Предположим, данное выражение вычисляется в следующем окружении:

(x := c? |=| d!5) || c!1

Тогда возможный эффект вычисления такого выражения состоит в выполнении взаимодействия посредством канала c, в результате чего получается выражение:

x := 1

В данном случае выбор в пользу первого выражения диктуется окружением (c!1).

Комбинатор внешнего выбора обладает свойствами коммутативности и ассоциативности.

Внутренний выбор

Внутренний (недетерминированный) выбор задается выражением вида:

$value\_expr_1$ |^| $value\_expr_2$,

где выражения $value\_expr_1$ и $value\_expr_2$ должны иметь одинаковый тип, который совпадает с типом всего выражения. В результате внутреннего выбора из двух выражений для вычисления выбирается только одно, причем в отличие от предыдущего случая окружение не оказывает никакого влияния на этот выбор. Символ |^| обозначает комбинатор внутреннего выбора.

Например, в выражении:

(x := c? |^| d!5) || c!1

выражение c!1 не влияет на то, какое из выражений x := c? или d!5 выбирается. Если внутренний выбор выпадает на x:=c?, то результатом является выражение:

x:= c? || c!1,

потенциально приводящее к взаимодействию через канал c. Если выбор выпадает на d!5, то результатом оказывается выражение:

d!5 || c!1,

препятствующее внутренним взаимодействиям.

Комбинатор внутреннего выбора также обладает свойствами коммутативности и ассоциативности.

Взаимная блокировка (interlock)

Взаимная блокировка (interlock) задается выражением вида:

$value\_expr_1$ ++ $value\_expr_2$,

где ++ обозначает комбинатор взаимной блокировки. Указанные выражения вычисляются параллельно, блокируя друг друга по отношению к внешним взаимодействиям, до тех пор, пока одно из них не завершится, в то время как другое продолжит свое вычисление. Оба выражения должны иметь тип **Unit**, этот же тип имеет и всё выражение. Отличие взаимной блокировки от параллельной композиции (||) состоит в том, что в течение параллельного вычисления не происходит взаимодействия с окружением, оно становится возможным только после снятия взаимной блокировки в результате завершения вычисления одного из выражений. Таким образом, взаимная блокировка вынуждает выражения взаимодействовать друг с другом.

Комбинатор взаимной блокировки обладает свойством коммутативности, но не является ассоциативным.

Рассмотрим несколько примеров в области видимости объявлений:

```
value e, e1, e2 : T
channel c , c1, c2: T
variable x : T
```

Отличие взаимной блокировки от параллельной композиции иллюстрируют две следующие эквивалентности:

```
x := c? ++ c!e is x := e
x := c? || c!e is (x := e) |^| ((x := c?; c!e)|=|(c!e; x := c?)|=|(x := e))
```

В первом случае наличие взаимной блокировки вынуждает выражения выполнить взаимодействие. Во втором случае в зависимости от внутреннего выбора выражений взаимодействие между ними либо происходит, либо окружение делает внешний выбор в пользу одного из указанных взаимодействий.

Взаимная блокировка наглядно иллюстрирует также различие между внешним и внутренним выбором:

```
(x := c1? |=| c2!e2) ++ c1!e1 is x := e1
(x := c1? |^| c2!e2) ++ c1!e1 is x := e1 |^| stop
```

При внутреннем выборе может произойти останов из-за невозможности выполнить взаимодействие (c2!e2 ++ c1!e1). Такая тупиковая ситуация специфицируется в RSL предопределенным выражением **stop**.

Другим примером того, как невозможность взаимодействия с окружением приводит к останову, является эквивалентность:

```
x := c1? ++ c2!e is stop
```

Для параллельной композиции соответствующая эквивалентность принимает вид:

x := c1? || c2!e **is** (x := c1?; c2!e) |=| (c2!e; x := c1?)

и выражает тот факт, что поскольку данные выражения не могут взаимодействовать друг с другом, для них возможно только взаимодействие с окружением.

Упрощение параллельных выражений

При упрощении параллельных выражений кроме указанных свойств полезно иметь в виду следующие эквивалентности:

value_expr$_1$ || (value_expr$_2$ |^| value_expr$_3$) **is**
(value_expr$_1$ || value_expr$_2$) |^| (value_expr$_1$ || value_expr$_3$)
value_expr$_1$ ++ (value_expr$_2$ |^| value_expr$_3$) **is**
(value_expr$_1$ ++ value_expr$_2$) |^| (value_expr$_1$ ++ value_expr$_3$)

Кроме того, если выражения value_expr, value_expr$_1$ и value_expr$_2$ не вызывают взаимодействия и c1 ~= c2, то справедливы свойства:

x := c1? || c2! value_expr **is** (x := c1?; c2! value_expr) |=| (c2! value_expr; x := c1?)
x := c? || c! value_expr **is** (x := value_expr) |^|
((x := c?; c! value_expr)|=|(c! value_expr; x := c?)|=|(x := value_expr))
x := c1? ++ c2! value_expr **is stop**
x := c? ++ c! value_expr **is** x := value_expr
(x := c1? |=| c2! value_expr$_2$) ++ c1! value_expr$_1$ **is** x := value_expr$_1$
(x := c1? |^| c2! value_expr$_2$) ++ c1! value_expr$_1$ **is** (x := value_expr$_1$) |^| **stop**

Упражнения

Упростить следующие параллельные выражения в предположении, что произошли все возможные взаимодействия:

1. (b!2) || (y := a?) || **if** (x := 1); (a!x); (**true** |^| **false**) **then** a!(b?) **else** a!(1 + b?) **end**

2. (b!2) || (x := a?) || **if** (a!3; **true**) |^| ((a!b? + 5); **false**) **then** a!(b?)
else a!(1 + b?) **end** || (a!0)

3. (**if** (a?) > 0 **then** b!1 **else** b!2 **end** ++(a!1; x := b?; b!4)) ||
(y := **if** (b?) = 1 **then** a? **else** (0 - a?) **end**) || (a!0)

4. a!(5 + b?) || ((x := (**if true** |^| **false then** x := b?; 1 **else** b!3; x := 2; 6 **end**) + x) ++
(b!4 || y := b?))

5. a!(5 + a?) || ((x := (**if true** |^| **false then** x := b?; 1 **else** x := b?; 6 **end**)) ++
(b!4 || x := a? || a!3 || y := b?))

6.
```
case (1 |^| b?) of
        1 -> x := a? + 1,
        2 -> x := b?,
        3 -> y := a? + 3,
        4 -> y := b? + a?,
        5 -> x := y := a?; y
   end || a!1 || b!(a? + 2) || a!3
```

7.
```
case (1 |^| b?) of
        1 -> x := a? + 1,
        2 -> x := b? |^| a?,
        3 -> y := a? + 3,
        4 -> y := b? + a?,
        5 -> x := y: = a?; y
   end || a!0 || b!(a? + a?) || a!2 || a!3
```

8.
```
case (a?) of
        0, 1 -> x := a? + 1,
        2 -> x := b?,
        3 -> y := a? + 3,
        4 -> y := b? + a?
   end || (x := a?; a!0 |=| b!(a?)) || a!4
```

ГЛАВА 10. НЕДЕТЕРМИНИЗМ И НЕПОЛНЫЕ СПЕЦИФИКАЦИИ

Понятие недетерминизма и неполной спецификации. Источники недетерминизма в спецификациях. Недетерминизм в **case**- и **let**-выражениях. Упражнения.

Понятие недетерминизма и неполной спецификации

Понятия недетерминизма и неполной спецификации имеют разную природу, но, вместе с тем, в спецификациях часто встречаются в похожих ситуациях, поэтому имеет смысл рассмотреть эти понятия совместно.

Сначала остановимся на *недетерминизме*, а потом перейдем к неполным спецификациям или, как говорят для краткости - *недоспецификациям*. Недетерминированным поведением мы называем такое поведение системы, когда две попытки выполнения одной и той же операции, в случае RSL одного и того же выражения, дают (могут дать) два разных результата. Сразу же сделаем одно важное замечание. В принципе, можно рассматривать недетерминизм реальной системы и недетерминизм ее модели, которая представлена некоторой спецификацией. В данной главе мы рассматриваем недетерминированные спецификации, при этом в общем случае возможны следующие сочетания реальных систем и их моделей:

- ДсДм - детерминированная система и детерминированная модель - это простейший случай;
- НсДм - недетерминированная система и детерминированная модель
- НсНм - недетерминированная система и недетерминированная модель
- ДсНм - детерминированная система и недетерминированная модель.

Пропуская случай ДсДм как достаточно простой, начнем сразу со второго случая. Вариант НсДм используется тогда, когда модель существенно упрощает поведение реальной системы и вместо рассмотрения разнообразных результатов/поведений, которые можно ожидать от реальной системы рассматривается один из возможных вариантов. Такие модели часто используются при прототипировании, когда важно сосредоточиться на одних вопросах функциональности и архитектуры и можно временно отвлечься от других. Заметим, для целей верификации такое соотношение системы модели может породить ряд трудностей, так, например, система, будучи недетерминированной, будет демонстрировать поведение, не предусмотренное моделью.

Необходимость случая НсНм также достаточно понятна - для системы со сложным, недетерминированным поведением приходится вводить в рассмотрение недетерминированную модель. При этом нужно иметь в виду, что наблюдаемое поведение системы и модели могут существенно различаться. В частности, на некоторых отрезках времени система может вести себя детерминировано, модель - недетерминировано, и - наоборот. Таким образом, соответствие поведений системы и модели в данном случае вопрос не простой.

Последний случай ДсНм кажется надуманным - не понятно, для чего моделировать детерминированную систему при помощи недетерминированной модели, то есть с поведением более разнообразным, чем поведение реализации. Но примеры такого сочетания тоже могут встретиться. Так рассмотрим систему, которая обрабатывает некоторые данные высокоточного вычислителя и выдает

округленный результат. В требованиях к системе написано, что округление можно проводить любым из способов, который реализован в использованном аппаратном процессоре, и это требование должно быть формализовано в спецификациях. Большинство процессоров реализуют либо округление к ближайшему целому, либо выполняют отбрасывание дробной части. В этих предположениях системы, которые будут выдавать в качестве результата ближайшее приближение исходных данных, не превышающее его, или не превосходящие его, или случайным образом выбирающие приближение сверху или снизу – все они отвечают требованиям модели. Пусть в спецификации не задается, каким именно образом выполняется округление и, если результат подходит под один из вариантов, описанных выше, он считается правильным. Тогда получается, что модель является недетерминированной, а в трех рассмотренных случаях только в последнем реализация недетерминирована, в первых двух она вполне детерминирована.

Неполная спецификация или недоспецификация, естественно может возникнуть на практике как результат ошибки ее разработчика или как результат его недобросовестного отношения к работе. Мы рассматриваем другую ситуацию, когда спецификация должна быть неполной. В каких случаях это бывает на практике? Имеется две основные причины, обуславливающих необходимость недоспецификации. Во-первых, это стремление к повышению уровня абстракции и нарочитом игнорировании деталей, чтобы дать разработчику реализации более широкое поле для поиска наилучших с его точки зрения решений. Во-вторых, причиной может быть отсутствие некоторых знаний в момент разработки спецификаций.

Пример. На ранней фазе проектирования уже принято решение, что размер пакетов данных будет ограничен некоторой константой, но величина этой константы еще не выбрана. Это дает основание ввести такую константу (имя константы) в формальную спецификацию, что, в частности, позволит вполне формально описывать некоторые свойства системы (например, длина элемента пакета не должна превышать максимальную длину пакета). Собственно ограничение на размер будет сформулировано в реализации. Заметим, что различные реализации при этом могут иметь различные ограничения на размер, но многие их свойства будут сформулированы в спецификациях в общем виде, не зависящем от деталей реализации.

Иногда недоспецификация и недетерминизм в формальном тексте на RSL выглядят очень похоже. Покажем на типовых примерах, что мы будем называть недоспецификацией и что – недетерминизмом.

Рассмотрим примеры определения недоспецифицированных констант и функций:

```
value   x : Int,
        y : Int :- y ~= 0
axiom  y ~= 3
```

Константа x может иметь любое целое значение. Константа y может иметь любое целое значение за исключением 0 и 3.

```
value f : Int -> Int
axiom forall x : Int :- f(x) > x
```

Функция f выдает результат, превосходящий значение аргумента. Заметим, что f описана как детерминированная функция. Это значит, что в ответ на вызов с одним и тем же аргументом она должна отвечать всегда одним и тем же результатом, конкретное значение которого пока не специфицировано.

Вот примеры недетерминированных функций:

```
value f : Int -~-> Int
axiom forall x :  Int :-
        (f(x) = x-1) \/
        (f(x) = x) \/
        (f(x) = x+1)
```

Функция f возвращает как результат целое значение, которое отличается от аргумента не более, чем на единицу. Функция недетерминирована, то есть при вызове с одним и тем же аргументом будет давать, возможно, разные результаты.

```
value   f : Int -~-> Int
        f(x) is    let y : Nat :- x-1 <= y /\
                                  y <= x+1
                   in y end
```

Это полностью эквивалентное определение функции f.

```
value f : Int -~-> Int
      f(x) is x-1 |^| x |^| x+1
```

Еще одно эквивалентное определение функции f, которое использует комбинатор *внутренний выбор*.

Заметим, что при использовании имплицитного (неявного) определения при помощи пост-условия функция определяется как детерминированная. Так, очень близкое по смыслу определение функции f1 будет определением детерминированной функции (при этом можно считать функцию недоспецифицированной):

```
value    f1 : Int -> Int
         f1(x) as r
  post  (r <= x+1) /\ (r >= x-1)
```

Источники недетерминизма в спецификациях

В какой форме недетерминизм может появиться в спецификациях? Перечислим основные возможности:

- выражения с использованием комбинатора внутренний выбор
- **let**-выражения в неявной форме
- особые случаи использования **case**- и **let**-выражения с record pattern
- выражения, использующие комбинатор внешний выбор и параллельные вычисления, вообще.

Кроме того, часто (но не всегда), если в выражении используется обращение к недетерминированной функции, то и само выражение становится недетерминированным.

Рассмотрим часто встречающиеся ошибки при использовании недетерминированных спецификаций. Первая из них – чисто техническая, забывают, что недетерминированная функция не может быть всюду вычислимой (total):

```
value f : Int -> Int      f(x) is 0 |^| 1 |^| 2
```

Правильно писать:

```
value f : Int -~-> Int      f(x) is 0 |^| 1 |^| 2
```

Вторая ошибка – попытка определения «недетерминированной константы», например:

```
value x : Int = 0 |^| 1 |^| 2
```

Value-expression в определениях констант и все предикаты в RSL – это детерминированные выражения.

Еще один важный случай, когда недетерминизм фактически запрещен – это предикаты, использующиеся как описания ограничений в аксиомах, пред- и пост-условиях, в других описаниях. Недетерминированный предикат тождественен **false**. Так, выражение:

```
axiom let b : Bool in b end
```

тождественно равно **false,** так как предикат, стоящий после слова **axiom** является недетерминированным. Следующий пример:

```
exists x : Char :- let y : Char in y end
```

Это выражение также равно **false,** так как ограничивающий предикат недетерминирован. Следовательно, выражение:

```
{ x | exists x : Char :- let y : Char in y end }
```

описывает пустое множество. Такие же правила использования предикатов справедливы в сокращенных описаниях списков и отображений, в описаниях подтипов и в имплицитной форме **let**-выражения.

Недетерминизм в case- и let-выражениях

Из перечисленных выше источников недетерминизма отдельного внимания заслуживает недетерминизм в **case**- и **let**-выражениях. Рассмотрим следующее определение множества, которое строится при помощи конструкторов empty и add:

```
type
    Set == empty | add(Elem, Set)
axiom forall e, e1, e2 : Elem, s : Set :-
      [no_duplicates]
          add(e, add(e,s)) is add(e,s)
      [unordered]
```

add(e_1, add(e_2,s)) **is** add(e_2, add(e_1,s))

Попробуем написать выражение, которое будет выбирать из значения типа Set некоторый (произвольный) элемент множества. В предположении, что

x – это переменная типа Set и
x ≠empty

таким выражением может быть либо

```
let add(x, y) = s in x end
```

либо

```
case s of
    add(x, y) -> x
end
```

Оба вышеприведенных выражения являются в общем случае недетерминированными, так как при данном определении Set , например, верно, что, если s= add(a, add(b, empty), то в силу аксиомы [unordered], верно, что s= add(b, add(a, empty). Следовательно, и в случае **let**, и **case** значение этих выражений может равняться либо a, либо b, то есть выражения – недетерминированы.

Упражнения

1. Описать недетерминированную функцию, которая выбирает из множества некоторый элемент.

2. Описать детерминированную функцию, которая выбирает из множества некоторый элемент. (При необходимости введите дополнительные ограничения на тип элементов).

3. Привести пример, когда комбинация двух недетерминированных функций дает детерминированный результат.

4. Какое из приведенных ниже выражений является недетерминированным?

   (a!1) || (b!1) || (x:=a?) || (x:=b?)
   ((a!1) || (b!1)) |=| ((x:=a?) || (x:=b?))
   (a!1) || (b!1) || ((x:=a?) |=| (x:=b?))

5. Привести пример **case**- и **let**-выражений, которые дают недетерминированный результат. Перепишите спецификацию таким образом, чтобы все возможные поведения были представлены в форме детерминированных выражений, объединенных комбинатором |^|.

ГЛАВА 11. ЗАДАНИЕ ПРАКТИКУМА

Постановка задачи. Варианты заданий. Требования и методические указания для выполнения задания.

Постановка задачи

Разработать на языке RSL спецификацию программного интерфейса определяемой вариантом задания системы в виде:

- явной моделе-ориентированной спецификации,
- неявной моделе-ориентированной спецификации,
- алгебраической спецификации.

При описании интерфейса предложенной системы предусмотреть в его составе операции по формированию информационной базы системы, обеспечивающей возможность накапливать и использовать необходимую информацию. Способ представления такой информационной базы выбрать самостоятельно. В предложенных ниже вариантах заданий дается лишь краткое описание программных систем с указанием минимально необходимого набора операций, тем самым разработчику спецификаций предоставляется возможность расширить этот набор по своему усмотрению.

Варианты заданий

1. Система учета ″автомобили – владельцы – доверенности″.
   Система должна обеспечивать следующие возможности: добавлять/удалять нового владельца и соответственно новый автомобиль для заданного владельца, осуществлять аналогичные операции с доверенностями на автомобиль, выдавать необходимую справочную информацию (например, для указанного автомобиля определять его владельца и т.д.), при этом предполагается, что у каждого автомобиля может быть только один владелец, на один и тот же автомобиль может быть зафиксировано несколько доверенностей.

2. Генеалогическое дерево.
   Система поддержки генеалогического дерева должна предоставлять следующие возможности: добавлять в дерево нового члена семьи (ребенка, супруга, предка), вносить изменения в узлы дерева (например, фиксировать смену фамилии или дату смерти), осуществлять поиск полезной информации по дереву (например, для указанного члена семьи находить его детей и наоборот).

3. Железнодорожное расписание станции.
   На железнодорожной станции имеется набор платформ и путей, на которые могут прибывать поезда, известны номера прибывающих поездов, время их прибытия и отправления, а также станции отправления и назначения. Система поддержки железнодорожного расписания станции должна обеспечивать возможность формирования расписания, внесения в него изменений и выдачу полезной информации (например, список поездов до указанной станции назначения).

4. Система поддержки составления расписания занятий.
   Система должна обеспечивать возможность составления расписания занятий для некоторого учебного заведения, внесения в расписание изменений и выдачу полезной информации (например, по итоговому расписанию получить расписание указанной группы на заданный день). В расписании должны фиксироваться время и место проведения занятия, предмет и преподаватель, проводящий занятие, а также номер группы, для которой это занятие проводится. Расписание не должно содержать коллизий (например, разные занятия не должны пересекаться друг с другом по месту и времени их проведения).

5. Планирование автобусного движения в районе.
   В районе имеется несколько автовокзалов, у каждого из которых есть ряд посадочных площадок. Между автовокзалами курсируют рейсовые автобусы, для каждого рейса фиксируется станция отправления и назначения, посадочная площадка, с которой происходит отправление, а также время посадки в автобус и время в пути. Система поддержки планирования автобусного движения должна обеспечивать возможность добавлять/удалять новые рейсы, вносить изменения в уже имеющиеся рейсы и выдавать полезную справочную информацию (например, для указанного автовокзала определять все рейсы, отправляющиеся с заданной посадочной площадки и т.д.).

6. Заказ и учет товаров в ″бакалейной лавке″.
   В ″бакалейной лавке″ для каждого товара фиксируется место хранения (определенная полка), количество и поставщик этого товара. Система поддержки заказа и учета товаров должна обеспечивать возможность добавления/удаления нового товара, изменения информации об имеющемся товаре (например, при изменении количества товара и т.д.) и выдачи необходимой справочной информации (например, список товаров, количество которых необходимо пополнить).

7. Управление библиотекой.
   В библиотеке осуществляется регистрация всех читателей и ведутся каталоги поступивших в библиотеку книг, кроме того для каждого читателя фиксируется информация о том, какие книги находятся у него на руках в данный момент. Система поддержки управления библиотекой должна обеспечивать возможность добавления/удаления читателей и соответственно книг в каталоги, регистрацию взятых и возвращенных читателем книг, а также выдавать полезную справочную информацию (например, о наличии в данный момент указанной книги и т.д.).

8. Управление памятью на диске.
   Система обслуживает запросы процессов на использование памяти. Процесс может запросить новую область памяти указанной длины, вернуть ее. При выделении области памяти по запросу процесса этот процесс становится владельцем данной области. Различные процессы могут получать доступ к

памяти на чтение и запись. Определение и изменение прав доступа к указанной области памяти осуществляется только ее владельцем. Неявная и алгебраическая спецификация должны описывать широкий спектр стратегий управления памятью.

9. Информационное табло по состоянию авиарейсов.
   На табло отражается следующая информация о рейсе: номер рейса, пункт вылета, время прилета по расписанию, ожидаемое время прилета, статус (отложен, вылетел, прилетел). Система поддержки информационного табло должна обеспечивать добавление и удаление информации о рейсах, а также внесение изменений в состояние табло, если произошло некоторое событие (например, вылет какого-то рейса отложен на N минут, произошла посадка самолета указанного рейса и т.д.)

10. Управление версиями программного проекта.
    Программный проект представляет собой некоторую совокупность программ, каждая программа в свою очередь состоит из файлов, файлы связаны друг с другом отношением ″использует″. Для каждого файла может существовать несколько его вариантов и для каждой программы соответственно несколько ее версий. Конкретный вариант файла однозначно идентифицируется именем файла и номером варианта, аналогично версия программы идентифицируется именем программы, номером версии и списком вариантов файлов. Каждая версия программы должна быть замкнута по отношению ″использует″. Система поддержки управления версиями должна обеспечивать возможность создавать новые варианты файлов и новые версии программ, добавлять и исключать варианты файлов из версий без нарушения указанной замкнутости, т.е. нельзя, например, удалить из версии какой-то вариант файла, если он используется оставшимися файлами.

11. Система поддержки составления расписания поездов на железной дороге.
    Железная дорога представляет собой сеть станций, связанных между собой путями, (причем могут существовать как однопутные, так и двухпутные перегоны), на каждой станции имеется N путей, для каждого перегона известно время, необходимое для его прохождения. В расписании указывается список поездов данной железной дороги с указанием маршрутов их следования (маршрут следования задается списком станций, на которых останавливается поезд) и временем прибытия/отправления на станции маршрута. Расписание должно быть свободно от коллизий, т.е. на пути перегона так же, как и на пути станции не может находиться одновременно более одного поезда. Система поддержки составления расписания должна обеспечивать возможность добавления и удаления новых маршрутов, внесения изменений в уже составленное расписание, а также выдачу полезной информации (например, по расписанию всей железной дороги получить расписание для указанной станции).

12. Управление процессами.
    Специфицируется набор операций, при помощи которых планировщик операционной системы поддерживает очереди процессов к ресурсам (например, к устройству ввода/вывода, памяти, процессору, порту ввода/вывода другого процесса), при этом внешние операции и/или прерывания, при помощи которых пользовательские процессы обращаются к планировщику (возбуждают требование на обслуживание планировщиком) не рассматриваются. Планировщик поддерживает следующие структуры данных: набор ресурсов, набор процессов, порты ввода/вывода процессов. Набор операций должен позволять строить и модифицировать перечисленные структуры данных, размещать процесс в очередях к ресурсам, изымать процесс из очереди.

13. Утилита make.
    Утилита получает на вход список файлов и функцию, которая для каждого файла выдает список файлов, от которых он зависит (например, по использованию типов данных). Кроме того утилита использует в качестве входных данных совокупность файлов. Все файлы, которые утилита получает как исходные данные (прямо или косвенно), должны принадлежать этой совокупности. Результатом утилиты должен стать список файлов – транзитивное замыкание исходного списка. Список-результат должен определять порядок обработки файлов: каждый «зависящий» файл не должен опережать в списке ни один из файлов, от которого он зависит.

14. Бронирование авиабилетов.
    В системе бронирования авиабилетов имеется набор авиарейсов, на которые можно бронировать билеты. Для каждого рейса фиксированы пункты отправления и назначения, а также дата и время вылета. Система должна обеспечивать возможность добавления/удаления рейсов, бронирования билетов /отказа от брони и выдавать полезную справочную информацию (например, об имеющихся свободных местах на рейс и т.д.)

15. Управление счетами в банке.
    В системе обеспечивается регистрация клиентов банка, каждый из которых может иметь в этом банке 0 или более счетов. Любой счет имеет неснижаемый остаток, по каждому счету ведется баланс. Система поддержки управления счетами в банке должна обеспечивать возможность добавления и удаления клиента банка, открытие и закрытие счета для клиента, просмотр баланса счета, снятие и добавление денег на счет, а также выдавать полезную справочную информацию (например, все счета данного клиента и т.д.).

16. Управление аптечной сетью.
    В аптечную сеть населенного пункта объединены несколько аптек. Все имеющиеся в аптеках препараты сгруппированы по категориям (жаропонижающие, болеутоляющие и т.д.), по каждой аптеке фиксируется информация о наличии или отсутствии в ней данного препарата. Система должна обеспечивать возможность формирования своей информационной базы (добавлять/удалять аптеку в сеть, добавлять/удалять соответствующие

наименования препаратов указанной категории) и выдавать справочную информацию полезного содержания (например, о наличии данного препарата в указанной аптеке или в сети аптек, все препараты указанной категории и т.д.)

17. Туристическое бюро.
    Справочная система по обслуживанию туристического бюро содержит информацию о загрантурах. Для каждого тура фиксируется место его проведения (страна, курорт, отель), категория отеля проживания, время и продолжительность тура, а также его стоимость и, возможно, некоторая дополнительная информация. Система должна обеспечивать возможность наполнения и изменения своей информационной базы – добавлять/удалять туры, вносить изменения в данные о туре (например, при изменении стоимости тура), а также выдавать полезную справочную информацию (поиск туров по курорту, в указанной ценовой категории, в заданном диапазоне времени и т.д.)

18. Бюро кредитных историй.
    Бюро кредитных историй обслуживает сеть банков. В бюро ведется регистрация клиентов, бравших кредиты в банках этой сети. По каждому кредиту фиксируется банк, выдавший кредит, размер кредита, а также состояние погашения кредита (кредит в процессе погашения, кредит погашен, невозвращенный кредит). В системе должна быть предусмотрена возможность расширения обслуживаемой сети за счет включения в нее нового банка. Кроме того система должна обеспечивать возможность добавления нового клиента и соответственно нового кредита к уже имеющемуся клиенту, внесения изменений о состоянии кредита (например, при погашении кредита), а также осуществлять поиск полезной справочной информации (например, находить всех клиентов с плохой/хорошей кредитной историей, все невозвращенные кредиты данного банка и т.д.).

Методические рекомендации

Разработка спецификации программного интерфейса системы должна начинаться с определения набора основных операций (функций), которые необходимо включить в состав интерфейса. Описание сигнатур этих функций остается фактически неизменным во всех трех видах спецификаций (явном, неявном, алгебраическом).

При разработке моделе-ориентированных спецификаций необходимо выбрать подходящую абстракцию данных и на ее основе промоделировать заданную программную систему. В качестве абстракции данных в зависимости от специфики решаемой задачи можно использовать имеющиеся в языке RSL абстракции множеств, списков и отображений. Набор операций должен содержать операции, позволяющие накапливать в системе необходимую информацию, т.е. осуществлять наполнение некоторой информационной базы системы, способ представления которой определяется выбранной абстракцией данных. Для этого обычно достаточно включить в состав операций следующие операции:

- инициализацию информационной базы пустым значением (в RSL это выражается путем определения соответствующей константы),

- добавление в базу нового элемента,
- удаление указанного элемента из базы.

При описании данных операций необходимо предусмотреть возможные ограничения на их выполнение, связанные с обеспечением непротиворечивости (консистентности) хранящейся в базе информации. Так, например, при формировании информационной базы системы поддержки составления расписания добавление нового занятия в расписание заданной группы может производиться только в том случае, если это не приведет к возникновению коллизий, т.е. если в указанное время у этой группы и преподавателя нет других занятий и свободна указанная аудитория. Одним из важных средств описания консистентного состояния являются инварианты, которые в RSL представляются либо в форме аксиом, либо в форме ограничений подтипов.

Весьма распространенным способом определения ограничения подтипа является использование для этой цели предиката-ограничения, представленного в виде функции is_wf_system(sys), с помощью которой проверяется сохранение инвариантов в моделируемой системе. В качестве примера рассмотрим моделирование базы данных, хранящей совокупность записей вида «ключ - данные», на основе абстракции множеств. Предикат is_wf_db(rs) задает ограничение подтипа, позволяющее из произвольных множеств записей указанного вида отбирать только те, которые удовлетворяют условию «хорошо сформированной» базы данных, т.е. гарантируют сохранение свойства уникальности ключа для всех записей в базе данных.

```
type
        Key, Data, Record = Key >< Data,
        DataBase={| rs : Record-set :- is_wf_db(rs)|}
value
        is_wf_db : Record-set -> Bool
        is_wf_db(rs) is (all k : Key, d1,d2 : Data :-
                                ((k,d1) isin rs /\ (k,d2) isin rs) => d1 = d2)
```

Кроме указанных основных операций по наполнению информационной базы системы полезно предусмотреть операции, позволяющие изменять отдельные элементы базы, а также осуществлять в базе поиск полезной с точки зрения разработчика спецификаций информации. Некоторые примеры таких операций приводятся в описании вариантов заданий практикума, однако их набор может быть расширен по желанию разработчика спецификаций.

При разработке неявных спецификаций функций следует обращать особое внимание на строгость формулировки постусловия, стараясь как можно тщательнее описать там результат выполнения функции и избежать по возможности возникновения неполной спецификации. Иллюстрацией неполной спецификации подобного рода может служить следующая спецификация функции добавления новой записи в описанную выше базу данных:

```
value
        add : Key >< Data >< DataBase -~-> DataBase
        add(k,d,db) as res
                post (k,d) isin db
        pre (k,d) ~ isin db
```

Предложенная спецификация является неполной, т.к. лишь частично описывает эффект от добавления нового элемента (k,d) в множество db. В частности, здесь игнорируется важный факт, что добавление элемента является единственным изменением, производимым функцией add над множеством db, и, следовательно, за исключением элемента (k,d) множества db и res должны полностью совпадать.

ГЛАВА 12. ПРИМЕРЫ ЭКЗАМЕНАЦИОННЫХ БИЛЕТОВ

Билет № 1.

1. Дано explicit определение функции. Написать implicit-спецификацию функции эквивалентной данной.

```
value
        f : Int >< Int >< Int -> write x read  y Int >< Int
        f(a,b,c) is       if a=b then
                                  (if a+b > c then c else a+b end,
                                   a*b*(if c>0 then x:=y; c else 0-c end))
                          else (a+b, x:=b; a-b)
                          end
```

Решение:

```
value
        f : Int >< Int >< Int -> write x read  y Int >< Int
        f(a,b,c) as (d, e)
        post
        if a=b then
                if a+b>c then
                          if c>0 then d=c /\ x=y /\ e = a*b*c
                          else d=c /\ x=x' /\ e=a*b*(0-c) end
                elseif c>0 then d=a+b /\ x=y /\ e= a*b*c
                else d=a+b /\ x=x‘ /\ e=a*b*(0-c) end
        else d=a+b /\ x=b /\ e=a-b
        end
```

2. Дать явное (explicit) или неявное (implicit) определение функций (включая слабейшие предусловия), отвечающих требованиям аксиом:

```
type S, E
value
create : Unit -> S,
add : E >< S -> S,
del : S -~-> S,
next : S  -~-> S,
get : S -~-> E,
append : S >< S -> S
```

```
axiom
all e : E, s : S :-
del( add( e, s ) ) is s,
all e : E, s : S :-
get( add( e, s ) ) is e,
all e : E, s : S :-
next( add( e, s )) is
        if s = create( ) then add( e, s )
        else append ( s, add( e, create( ) )
        end,
all e : E, s1, s2 : S :-
append( create(), s2 )) is s2,
all e : E, s1, s2 : S :-
append( add(e, s1), s2 )) is
add (e, append (s1, s2))
```

Решение:

```
type S, E
value
create : Unit -> S
create is <..>,
```

```
add : E >< S -> S
add (e,s) is <.e.> ^ s,

del : S -~-> S
del (s) is tl s
pre s~=<..>,

next : S  -~-> S
next (s) is tl s ^ <. hd s  .>
pre s~= <..>,

get : S -~-> E
get (s) is hd s
pre s ~= <..>,

append : S >< S -> S
append (s1, s2) as s
post
        if s1=<..> then s=s2
        else s=add(hd s1, append( tl s1, s2))
        end
```

3. Доказать, что правая спецификация является (или не является) уточнением левой.

value
f1 : M >< M -> M,
f2 : M -> S >< S,
f3 : E >< S -> **Bool,**
f4 : E >< E -> M
axiom
  all a1, a2, b1, b2 : E :-
  let (p,q) = f2(f1(f4 (a1,b1), f4(a2,b2)))
in
  f3(a1,p) /\ f3(b1,q) /\ f3(a2,p) /\ f3(b2,q)
end is true
	type
	  M = E –m-> E,
	  S = E**-set**
	value
	f1 : M >< M -> M
	f1(m1,m2) **is** m1 **!!** m2,
	f2 : M -> S >< S
	f2(m) **is** (**dom** m, **rng** m),
	f3 : E >< S -> **Bool**
	f3(a,b) **is** a **isin** b,
	f4 : E >< E -> M
	f4(a,b) **is** [a +> b]

Решение:

[[раскрыть all]]
let (p, q) = f2(f1(f4(a1, b1), f4(a2, b2))) **in**
f3(a1, p) /\ f3(b1, q) /\ f3(a2, p) /\ f3(b2,q) **end**
is true

[[раскрыть f4]]
let (p, q) = f2(f1([a1 +>b1], [a2+>b2])) **in**
f3(a1, p) /\ f3(b1, q) /\ f3(a2, p) /\ f3(b2,q) **end**
is true

[[раскрыть f1]]
let (p, q) = f2([a1 +>b1, a2+>b2]) **in**
f3(a1, p) /\ f3(b1, q) /\ f3(a2, p) /\ f3(b2,q) **end**
is true

[[раскрыть f2]]
let (p, q) = **if** a1=a2 **then** ({a1, a2}, {b2}) **else** ({a1, a2}, {b1, b2}) **end in**
f3(a1, p) /\ f3(b1, q) /\ f3(a2, p) /\ f3(b2,q) **end**
is true

[[раскрыть f3]]
let (p, q) = **if** a1=a2 **then** ({a1, a2}, {b2}) **else** ({a1, a2}, {b1, b2}) **end in**
a1 **isin** p /\ b1**isin** q /\ a2 **isin** p /\ b2 **isin** q **end**
is true

[[вычислить]]
let (p, q) = ({a1, a2}, **if** a1=a2 **then** {b2} **else** {b1, b2} **end in**
a1 **isin** p /\ b1**isin** q /\ a2 **isin** p /\ b2 **isin** q **end**
is true

[[раскрыть let]]
a1 **isin** {a1, a2} /\ b1 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end** /\ a2 **isin** {a1, a2} /\ b2 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end end**
is true

[[вычислить]]
true /\ b1 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end** /\ **true** /\ b2 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end end**
is true

[[упростить /\ ]]
b1 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end** /\ b2 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end end**
is true

[[вычислить if]]
b1 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end** /\ **true**
is true

[[упростить /\ ]]
b1 **isin if** a1=a2 **then** {b2} **else** {b1, b2} **end**
is true

[[анализ case]]
[[non_eq]] ~a1=a2 [[eq]] a1=a2

[[предположение non_eq]] ~a1=a2
b1 **isin** {b1, b2}
is true

[[вычислить]]
true
is true

true

[[предположение eq]] a1=a2
b1 **isin** {b2}
is true

[[вычислить]]

false
is true

false

Ответ: Нет, правая спецификация не является уточнением левой.

4. Дано определение вариантного типа:
type Elem, Collection == empty | add (Collection <-> head, first:Elem, Elem <-> new_second)

Дать эквивалентное определение без использования вариантных определений.

Решение:

```
type Elem, Collection
value  empty : Collection,
       add : Collection >< Elem >< Elem -> Collection,
       head : Collection >< Collection -~-> Collection,
       first : Collection -~-> Elem,
       new_second : Elem >< Collection -~-> Collection

axiom
[disjoint]
all c,c1 ; Collection, e1, e2 ; Elem :-
empty ~= add(c, e1, e2)

[induction]
all isCollection ; Collection -> Bool :-
       isCollection(empty) /\ (all c1 : Collection, e1, e2 ; Elem :- (isCollection(c1) =>
                                    isCollection(add(c1, e1, e2))))  =>
                                          (all c2 : Collection :- isCollection ( c2 ))

[head_add]
all c1, c2 ; Collection, e1, e2 : Elem :- head (c1, add(c2, e1, e2)) is add(c1, e1, e2))

[first_add]
all c: Collection, e1, e2 : Elem :- first (add(c, e1, e2)) is e1

[new_second_add]
all c: Collection, e1, e2, e3 : Elem :- new_second (e1, add(e2, e3)) is add(c, e2,e1)
```

Билет № 2.

1. Дано explicit определение функции. Написать implicit-спецификацию функции эквивалентной данной.

```
value
        f : Int >< Int -> write x read y Int >< Int >< Int
        f (a, b) is
          local variable v : Int := 0 in
                x:=0;
                for i in <.a..b.> do
                        x:=x+y; v := v+2*i
                end; (a,b,v+y)
          end
```

Решение:

```
value
        f : Int >< Int -> write x read y Int >< Int >< Int
        f (a, b) as (c, d, e)
        post
                a = c /\ b = d /\
                if a > b then e = y /\ x = 0
                else x = y * (b-a+1) /\ e = (a+b)*(b-a+1)+y
                end
```

2. Дать explicit или implicit определение функций (включая слабейшие предусловия), отвечающее требованиям аксиом:

```
type S, E
value
create : Unit -> S,
push_down : E >< S -> S,
push_right : E >< S -> S,
pop_up : S -~-> S,
pop_left : S  -~-> S,
get : S -~-> E,
```

```
axiom
        all e : E, s : S :-
        pop_up( push_down( e, s ) ) is s,

        all e : E, s : S :-
        pop_up( push_right( e, s ) ) is pop_up( s ),

        all e : E, s : S :-
        get( push_down( e, s ) ) is e,

        all e : E, s : S :-
        get( push_right( e, s ) ) is e,

        all e : E, s : S :-
        pop_left( push_right( e, s ) ) is s
```

Решение:

```
type S = L-list, L = E-list, E
value
create : Unit -> S
create () is <..>,
push_down : E >< S -> S
push_down (e,s) is <. <. E .> .> ^ S,
push_right : E >< S -> S
push_right (e,s) is     if s = <..> then <. <. e .> .>
                        else <. <. e .> ^ hd s .> ^ tl s
                        end,
pop_up S -~-> S
```

```
pop_up (s) is tl s
pre s ~= <..>,

pop_left : S -~-> S
pop_left (s) is <. tl hd s .> ^ tl s
pre s~= <..> /\ hd s~=<..>,

get : S -~-> E
get (s) is hd hd s
pre s~ ~= <..> /\ hd s~ ~= <..>
```

3. Рассмотреть возможные варианты результатов вычислений в предположении, что все возможные обмены сообщениями произошли

```
(       ( (x:=a?+1) || ( x:=b?) )
        |=|
        ( (x:=c?+1) )
        |=|
        (y:=b?+a?)  )
|| a!0 || b!2
```

Решение:

```
((x:=1) || (x:=2)) |^| (y:=2)
```

4. Дано определение вариантного типа:

```
type    Collection == empty | atom (head:Elem) | list(head : Collection), Elem
```

Дать эквивалентное определение без использования вариантных определений.

Решение:

```
type Elem, Collection
value  empty : Collection,
       atom : Elem -> Collection,
       list : Collection -> Collection,
       head : Collection -~-> Collection,
       head : Collection -~-> Elem

axiom
[disjoint]
all c ; Collection, e ; Elem :-
empty ~= atom(e) /\ empty~=list(c) /\ atom ( e ) ~= list ( c)

[induction]
all Elem, isCollection ; Collection -> Bool :-
isCollection(empty) /\ (all c : Collection, e ; Elem :-
                        isCollection (atom(e)) /\
                        (isCollection(c) => isCollection(list(c)) )) =>
                                (all c : Collection :- isCollection ( c ))

[head_list]
all c : Collection :- head (list(c)) is c

[head_atom]
```

all e : Elem :- head (atom(e)) **is** e

ОТВЕТЫ И РЕШЕНИЯ К УПРАЖНЕНИЯМ

1.1. (b) **type** ST = {| n : **Int**:- n \ 2 = 1 |}

1.2. (a) да
(b) да
(c) нет
(d) нет
(e) да
(f) нет
(g) нет
(h) да
(i) да
(j) нет
(k) да

1.3. (a) **false** (по свойству (1))
(b) a
Решение: **if** a **then ~(a is chaos) else false end** ≡ (по свойству (3))
if a **then ~(true is chaos) else false end** ≡
if a **then true else false end** ≡ (по свойству (3))
if a **then a else a end** ≡ **a**

1.4. (a) да (для данного i выбираем $j = -i$)
(b) нет (не верно, например, для $i > 0$)
(c) нет

1.5. (a) **all** i **: Int** :- **exists** j **: Int :-** j > i или ~(**exists** i **: Int :- all** j **: Int** :- i **>=** j)
(d) **exists** k **: Nat** :-n = k * k

2.1. (a) **type** Points = {| (x, y) : **Real >< Real** :- x >= 0 /\ y >= 0 /\ y >= x |}

2.2. (a) is_even **: Nat** -> **Bool**
is_even(n) **is** (**exists** m **: Nat** :- n = 2 * m)
или альтернативное определение:
is_even **: Nat** -> **Bool**
is_even(n) **is** n \ 2 = 0

2.3. (a) **value**
max **: Int >< Int** -> **Int**
max(i, j) **is if** i >= j **then** i **else** j **end**
(b) **value**
max **: Int >< Int** -> **Int**
max(i, j) **as** y
post (y = i /\ y >= j) \/ (y = j /\ y >= i)
(c) **value**
max **: Int >< Int** -> **Int**
axiom
all i,j **: Int** :- max(i, j) **is if** i >= j **then** i **else** j **end**

2.4. (a) **type** Complex = **Real >< Real**
(b) **value** zero **:** Complex = (0.0,0.0)
(c) **value** c **:** Complex :- **let** (x, y) = c **in** x = y **end**

(d) **value**
add **:** Complex >< Complex **->** Complex
add((x₁,y₁), (x₂,y₂)) **is** (x₁ + x₂, y₁ + y₂),
mult **:** Complex >< Complex **->** Complex
mult((x₁,y₁), (x₂,y₂)) **is** (x₁ * x₂ – y₁ * y₂, x₁ * y₂ + y₁ * x₂)

(e) **value**
f : Complex **->** Complex
f(c₁) **as** c₂ **post** c₁ ~= c₂

2.5. (a) **type**
Circle = Center >< Radius,
Center = Position,
Radius = {| r **: Real** :- r >= 0.0 |}

(b) **value**
on_circle **:** Circle >< Position -> **Bool**
on_circle((c,r), p) **is** distance(c, p) = r

или альтернативное определение:

value
on_circle **:** Circle >< Position -> **Bool**
on_circle(circ, p) **is**
let
(c,r) = circ
in
distance(c,p) = r
end

(c) **value**
circle **:** Circle = (origin, 3.0)

(d) **value**
pos **:** Position :- on_circle(circle, pos)

2.7. **value**
approx_sqrt **: Real** >< **Real -**~-> **Real**
approx_sqrt(x, eps) **as** s
post s ** 2.0 <= x /\ x < (s + eps) ** 2.0
pre x >= 0.0 /\ eps > 0.0

2.8. (b) **value**
F2 **: Nat** >< **Nat** >< **Nat** -> **Nat** >< **Nat**
F2(a, b, c) **as** (v, w)
post
if a + b > c **then** v = c **else** v = a + b **end** /\
if c > 0 **then** w = a * b * c **else** w = a * b * (0 –c) **end**

3.1. (a) {1, 3, 5, 7, 9} или {s | s : **Nat** :- (s \ 2 = 1) /\ (s < 10) }
или {2 * n + 1 | n : **Nat** :- n <= 4 }

(b) {2, 3, 5, 7, 11, 13, 17, 19} или
{n | n : **Nat** :- is_a_prime(n) /\ n < 20 },
is_a_prime : **Nat** -> **Bool**
is_a_prime(n) is (**all** k : **Nat** :- (k < n /\ k > 1) => n \ k ~= 0)

3.2. **card** { n | n **: Nat** :- 30 \ n = 0} = **card** {1, 2, 3, 5, 6, 10, 15, 30} = 8

3.3. **card** { (x, y) | x, y : **Int** :- x * x + y * y < 25 }

3.4. (a)

```
value
    max : Int-set -~-> Int
    max(s) as m
    post
        m isin s /\ (all k : Int :- k isin s => k <= m)
    pre s ~= { }
```

3.5. (a)

```
scheme
  UNIVERSITY_SYSTEM =
    class
      type
        Student,
        Course,
        CourseInfo = Course >< Student-set,
        University = {| (ss, cis) : Student-set >< CourseInfo-set :- is_wf(ss, cis) |}
      value
        /* is_wf осуществляет проверку на «хорошо сформированную» базу данных,
           проверяя выполнение свойства уникальности курса */
        is_wf :- Student-set >< CourseInfo-set -> Bool
        is_wf(ss, cis) is (all (c, ss1), (c1, ss2) : CourseInfo :-
                (c, ss1) isin cis /\ (c1, ss2) isin cis => (c = c1 = > ss1 = ss2)),
        /* hasStudent проверяет, учится ли данный студент в указанном
           университете */
        hasStudent : Student >< University -> Bool
        hasStudent(s, (ss, cis)) is s isin ss,
        /* hasCourse проверяет, читается ли данный курс в указанном университете
           */
        hasCourse : Course >< University -> Bool
        hasCourse(c, (ss, cis)) is
                (exists (c1, ss1) : CourseInfo :- (c1, ss1) isin cis /\ c = c1),
        /* studOf возвращает множество студентов заданного университета,
           посещающих указанный курс */
        studOf : Course >< University -~-> Student-set
        studOf(c, (ss, cis)) is
            {s | s : Student :- exists ss1 : Student-set :- s isin ss1 /\ (c, ss1) isin cis}
        pre hasCourse(c, (ss, cis)),
        /* attending возвращает множество курсов, которые посещает данный
           студент в указанном университете */
        attending : Student >< University -~-> Course-set
        attending(s, (ss, cis)) is
            {c | c : Course :- exists ss1 : Student-set :- (c, ss1) isin cis /\ s isin ss1}
        pre hasStudent(s, (ss, cis)),
        /* newStud добавляет нового студента к числу студентов заданного
           университета */
        newStud : Student >< University -~-> University
        newStud(s, (ss, cis)) is (ss union {s}, cis)
        pre ~hasStudent(s, (ss, cis)),
        /* dropStud исключает указанного студента из числа студентов заданного
           университета */
        dropStud : Student >< University -~-> University
        dropStud(s, (ss, cis)) is
            (ss \ {s}, {(c, ss1 \ {s}) | (c, ss1) : CourseInfo :- (c, ss1) isin cis})
        pre hasStudent(s, (ss, cis)),
```

```
        /* newCourse добавляет новый курс с пустым множеством студентов к числу
           курсов заданного университета */
        newCourse : Course >< University -~-> University
        newCourse(c, (ss, cis)) is (ss, cis union {(c, {})})
        pre ~hasCourse(c, (ss, cis)),
        /* delCourse удаляет указанный курс из числа курсов заданного
           университета */
        delCourse : Course >< University -~-> University
        delCourse(c, (ss, cis)) is (ss, cis \ {(c, ss1) | ss1 : Student-set :- true})
        pre hasCourse(c, (ss, cis))
    end
(b) value
        studOf : Course >< University -~-> Student-set
        studOf(c, (ss, cis)) as ss1
        post (c, ss1) isin cis
        pre hasCourse(c, (ss, cis))
```

4.1 (a) 169
(b) **chaos**

4.2. (a) **<.**2,4,6,8,10,12,14,16,**.>** или **<.**2 * n | n **in <.**1**..**8**.>.>**
(b) **<.**1,4,9,16,25,36,49,64,81,100**.>** или
<.n | n **in <.**1**..**100**.> :- exists** k **: Nat** :-n = k * k**.>**

4.3.
```
value
    Fib_numbers : Nat-inflist
axiom
    Fib_numbers(1) = 1,
    Fib_numbers(2) = 1,
    all i : Nat :- i > 2 =>
        Fib_numbers(i) = Fib_numbers(i - 2) + Fib_numbers(i - 2)
```
Искомый список **<.**n | n **in** Fib_numbers **:-** n <= 1000**.>**

4.4.
```
value
    is_sorted : Int-list -> Bool
axiom
    all L : Int-list :-
        is_sorted(L) is
            (all i1, i2 : Nat :- {i1, i2} <<= inds L /\ i1 < i2 => L(i1) < L(i2))
```

4.5.
```
type Elem
value
 (a) length : Elem-list -> Nat
        length(k) is if k = <..> then 0 else 1 + length(tl k) end
        или
     length : Elem-list -> Nat
        length(k) is card (inds k)
        или
     length : Elem-list -> Nat
        length(k) is
            case k of
                <..> -> 0,
                <.i.> ^ lr -> length(lr) + 1
            end
 (b) rev : Elem-list -> Elem-list
```

```
rev(k) is if k = <..> then <..> else rev(tl k) ^ <.hd k.> end
```

или

```
rev : Elem-list -> Elem-list
rev(k) is <.k(len k – i + 1) | i in <.1 .. len k.>.>
```

(c)
```
del : Elem-list >< Elem-> Elem-list
del(k, e) is
    case k of
      <..> -> <..>,
      <.i.> ^ lr -> if i = e then del(lr, e)
                              else <.i.> ^ del(lr, e) end
    end
```

или

```
del : Elem-list >< Elem-> Elem-list
del(k, e) is <.x | x in k :- x ~= e.>
```

(d)
```
number_of : Elem-list >< Elem-> Nat
number_of(k, e) is card {i | i : Nat :- i isin inds k /\ k(i) = e}
```

4.7.
```
scheme
  PAGE =
    class
      type Page = Line-list, Line = Word-list, Word
      value
        is_on : Word >< Page -> Bool
        is_on(w, p) is (exists i : Nat :- i isin inds p /\ w isin elems p(i)),
        number_of : Word >< Page -> Nat
        number_of(w, p) is
          card {(i, j) | i,j : Nat :- i isin inds p /\ j isin inds p(i) /\ w = p(i)(j)}
    end
```

5.1. (b)
```
[(n, k) +> n \ k | n, k : Nat :- (n <= 50) /\ (k > 1) /\ (k < n)]
```

(c)
```
[n +> {k | k : Nat :- n \ k = 0 /\ k <= n /\ k >= 1} | n : Nat :- n <= 20]
```

(d)
```
[n +> m | n, m : Nat :- exists k : Nat :-
    k * k = m /\ m <= n /\ (k + 1) * (k + 1) > n]
```

5.2.
```
scheme
MAP_UNIVERSITY_SYSTEM =
  class
    type
      Student,
      Course,
      CourseInfos = Course -m-> Student-set,
      University = Student-set >< CourseInfos
    value
      /* hasStudent проверяет, учится ли данный студент в указанном
        университете */
      hasStudent : Student >< University -> Bool
      hasStudent(s, (ss, cis)) is s isin ss,
      /* hasCourse проверяет, читается ли данный курс в указанном университете
        */
      hasCourse : Course >< University -> Bool
      hasCourse(c, (ss, cis)) is c isin dom cis,
      /* studOf возвращает множество студентов заданного университета,
        посещающих указанный курс */
      studOf : Course >< University -~-> Student-set
      studOf(c, (ss, cis)) is cis(c)
```

```
      pre hasCourse(c, (ss, cis)),
      /* attending возвращает множество курсов, которые посещает данный
        студент в указанном университете */
      attending : Student >< University -~-> Course-set
      attending(s, (ss, cis)) is
          {c | c : Course :- c isin dom cis /\ s isin cis(c)}
      pre hasStudent(s, (ss, cis)),
      /* newStud добавляет нового студента к числу студентов заданного
        университета */
      newStud : Student >< University -~-> University
      newStud(s, (ss, cis)) is (ss union {s}, cis)
      pre ~hasStudent(s, (ss, cis)),
      /* dropStud исключает указанного студента из числа студентов заданного
        университета */
      dropStud : Student >< University -~-> University
      dropStud(s, (ss, cis)) is
          (ss \ {s}, [ c +> cis(c) \ {s} | c : Course :- c isin dom cis ])
      pre hasStudent(s, (ss, cis)),
      /* newCourse добавляет новый курс с пустым множеством студентов к числу
        курсов заданного университета */
      newCourse : Course >< University -~-> University
      newCourse(c, (ss, cis)) is (ss, cis !! [c +> {}])
      pre ~hasCourse(c, (ss, cis)),
      /* delCourse удаляет указанный курс из числа курсов заданного
        университета */
      delCourse : Course >< University -~-> University
      delCourse(c, (ss, cis)) is (ss, cis \ {c})
      pre hasCourse(c, (ss, cis))
  end
```

6.1.

```
  scheme
  STACK_ALG =
    class
      type Elem, Stack
      value
        empty : Stack,
        push : Elem >< Stack -> Stack,
        pop : Stack -~- Stack,
        is_empty : Stack -> Bool,
        top : Stack -~-> Elem
      axiom
        [ is_empty_empty ]
            is_empty(empty) is true,
        [ is_empty_push ]
        all e : Elem, s : Stack :-
            is_empty(push(e,s)) is false,
        [ top_push ]
        all e : Elem, s : Stack :-
            top(push(e,s)) is e,
        [ pop_push ]
        all e : Elem, s : Stack :-
            pop(push(e,s)) is s
    end
```

6.3.
```
scheme
STACK_LIST =
  class
    type Elem, Stack = Elem-list
    value
      empty : Stack = <..>,
      push : Elem >< Stack -> Stack
      push(e,s) is <. e .> ^ s,
      pop : Stack -~- Stack
      pop(s) is tl s
          pre s ~= empty,
      is_empty : Stack -> Bool
      is_empty(s) is s = empty,
      top : Stack -~-> Elem
      top(s) is hd s
          pre s ~= empty
  end
```

Данная схема является уточнением схемы STACK_ALG, т.к. в ней определены все типы и функции из схемы STACK_ALG и справедливы все аксиомы исходной схемы. В качестве примера приведем доказательство справедливости аксиомы [top_push] для схемы STACK_LIST (справедливость остальных аксиом доказывается аналогично).

```
[ top_push ]
    all e : Elem, s : Stack :-
        top(push(e,s)) is e
[[раскрыть квантор]]
    top(push(e,s)) is e
[[раскрыть идентификатор push]]
    top(<. e .> ^ s) is e
[[раскрыть идентификатор top]]
    hd (<. e .> ^ s) is e
[[вычислить hd]]
    e is e
[[тождественное преобразование is]]
    true
```

6.5. Да, вторая спецификация является уточнением первой. В качестве примера приведем доказательство справедливости третьей аксиомы (справедливость остальных аксиом доказывается аналогично).

```
all a : A, b : L :-
    f1(a,b) is f2(a,b)
    pre f3(b) = 1
[[раскрыть квантор]]
    if f3(b) = 1 then f1(a,b) is f2(a,b) else true end
[[раскрыть идентификаторы f3,f1,f2]]
    if len b =1 then <. a .> ^ tl b is tl b ^ <. a .> else true end
[[вычислить tl при len b =1]]
    if len b =1 then <. a .> ^ <..> is <..> ^ <. a .> else true end
[[вычислить ^]]
    if len b =1 then <. a .> is <. a .> else true end
[[тождественное преобразование is]]
    if len b =1 then true else true end
```

```
[[вычислить if]]
    true
```

7.1. **type** Record = = new_record(key_of : Key, data_of: Data)

Эквивалентное определение:

```
type Record
value
    new_record : Key >< Data -> Record,
    key_of : Record -> Key,
    data_of : Record -> Data
axiom
    all p : Record -> Bool :-
            (all (k,d) : Key >< Data :-p(new_record(k,d))) => (all r : Record :- p(r)),
    all k : Key, d : Data :- key_of(new_record(k,d)) = k,
    all k : Key, d : Data :- data_of(new_record(k,d)) = d
```

7.2. **type** Stack = = empty | push(top : Elem, pop : Stack)

7.3. (b)

```
type Tree
value
    empty : Tree,
    node : Tree >< Elem >< Tree -> Tree,
    left : Tree -~-> Tree,
    val : Tree -~-> Elem,
    rigth : Tree -~-> Tree
axiom
    [disjoint]
        all t1,t2 :Tree, e : Elem:- empty ~= node(t1,e,t2),
    [induction]
        all p : Tree -> Bool :-
            (p(empty) /\
            (all t1,t2 : Tree, e : Elem:- p(t1) /\ p(t2) => p(node(t1,e,t2)))) =>
                    (all t : Tree :- p(t)),
    [left_node]
        all t1,t2 : Tree, e : Elem:- left(node(t1,e,t2)) is t1,
    [val_node]
        all t1,t2 : Tree, e : Elem:- val(node(t1,e,t2)) is e,
    [right_node]
        all t1,t2 : Tree, e : Elem:- right(node(t1,e,t2)) is t2
```

8.1. (a) x := 2
(b) x := 2; 3
(c) x := 1; 3

8.2. (a)

```
scheme I_STACK_EX =
    class
        type Elem
        variable st : Elem-list
        value
            empty : Unit -> write st Unit
            empty() is st := <..>,
            push : Elem -> write st Unit
            push(e) is st := <. e .> ^ st,
            pop : Unit -~-> write st Unit
```

```
            pop() is st := tl st
                pre st ~= <..>,
            is_empty : Unit -> read st Bool
            is_empty() is st = <..>,
            top : Unit -~-> read st Elem
            top() is hd st
                pre st ~= <..>
        end
```

8.2. (b)

```
scheme I_STACK_IM =
    class
        type Elem
        variable st : Elem-list
        value
            empty : Unit -> write st Unit
            empty() post st = <..>,
            push : Elem -> write st Unit
            push(e) post st = <. e .> ^ st',
            pop : Unit -~-> write st Unit
            pop() post st = tl st'
                pre st ~= <..>,
            is_empty : Unit -> read st Bool
            is_empty() as b post b = (st = <..>),
            top : Unit -~-> read st Elem
            top() as e post e = hd st
                pre st ~= <..>
    end
```

8.3. (a)

```
variable result : Real
value
    fract_sum : Nat -~-> write result Unit
    fract_sum(n) is
        local variable counter : Nat := n in
            result := 0.0;
            while counter> 0 do
                result := result + 1.0 / (real counter);
                counter := counter – 1
            end
        end
      pre n > 0
```

8.4. (a)

```
value
  f : Int >< Int >< Int -> write x, y Int >< Int
      f(a,b,c) as (d, e)
        post
          if a = b then
                    y = y' /\ (if a + b > c then d = c else x = y /\ d = a + b end) /\
                    (if c > 0 then x =c /\ d = b * c else x = x' /\ d = b * (0 – c) end)
                else (x = x' /\ y =a + b /\ d = a + b /\ e = a - b) end
```

9.1.

```
x := 1; (y := 1) ; (a!2) |^|
x := 1; (y := 1) ; (a!3)
```

9.2.

```
x := 3 || a!2 || a!0 |^|
x := 0 || a!3 ; a!2 |^|
x := 7 ; a!(1 + b?) || a!0 |^|
```

x := 0 || a!7; a!(1 + b?)

9.3. x := 1; y := 0

9.6. x := 2 || b!5 |^|
x := 4 || b!3 |^|
y := 1; x := 1 |^|
y := 6

ЛИТЕРАТУРА

1. The RAISE specification language. Prentice Hall, 1992.
2. RAISE Tools Reference Manual. LACOS/CRI/DOC/13/1/V2, 1994.
3 Е.А. Кузьменкова, А.К. Петренко. Формальная спецификация программ на языке RSL (методическое пособие по практикуму), М., Изд-во АО "Диалог-МГУ", 1999.
4 Е.А. Кузьменкова, А.К. Петренко. Формальная спецификация программ на языке RSL (конспект лекций), М., Издательский отдел факультета ВМиК МГУ, 2001.
5 ftp://ftp.iist.unu.edu/pub/RAISE/course_material

СОДЕРЖАНИЕ