Об одном примере нарушения принципа подстановки Лисков

А.Г.Пискунов, С.М.Петренко

5 января 2012 г.

АННОТАЦИЯ

В статье используется прием RAISE Development Method-а от группы авторов, называемой далее RDG (RAISE Development Group), для анализа примера Роберта Мартина, иллюстрирующего принцип подстановки (или иначе принцип замещения) Барбары Лисков. Данный технический прием алгебраического проектирования (впервые упомянутый Гуттагом) позволил уточнить понятия типа, выделения подтипа и взаимосвязь выделения подтипа и наследования.

Содержание

1 ВВЕДЕНИЕ

Принцип подстановки Лисков помогает понять суть термина suptyping - выделение подтипа, а статья Роберта Мартина [8] показывает некоторое несоответствие между наследованием в языке C++ и выделением подтипа. В статье Мартина принцип формулируется следующим образом:

```
If for each object o1 of type S there is an object
o2 of type T such that for all programs P defined
in terms of T, the behavior of P is unchanged when o1
is substituted for o2 then S is a subtype of T.
```

Перевод может звучать как - то так

```
Если для каждого объекта o1 типа S существует объект o2
типа T и любая программа P, записанная в терминах T,
не изменяет своего поведения при подстановке объекта o1
вместо объекта o2, то тип S является подтипом T.
```

Формулировка допускает достаточно неоднозначную трактовку. Чтобы понять, что имеется ввиду, рассмотрим пример Мартина с наследованием класса квадратов из класса прямоугольников.

Определим родительский класс *Rectangle*

```
---- File:./rsl/rectangle.cpp


class Rectangle{
  public:
    virtual void SetWidth(double w)  {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
    double GetHeight() const {return itsHeight;}
    double GetWidth()  const {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};

---- End Of File:./rsl/rectangle.cpp
```

и наследуем из него класс *Square*

```
---- File:./rsl/square.cpp
```

```
class Square : Rectangle {
  public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};
void Square::SetWidth(double w){
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h){
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}

---- End Of File:./rsl/square.cpp
```

Далее, автор статьи не без основания утверждает, что в приведенным примере в функции $LSPV$ таки нарушается принцип подстановки:

```
---- File:./rsl/violation.cpp


void LSPV(Rectangle& r){
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}

---- End Of File:./rsl/violation.cpp
```

Вызывает возражения не слишком хорошая формулировка: расплывчатость термина 'подстановки'('замещения') (хотя в данном примере это понятно, в программу LSPV передается ссылка на объект подкласса), непонятность термина 'поведение программы' и, соответственно, 'изменение поведения программы' и так далее.

Попробуем проанализировать пример с точки зрения методов работы RDG [4] и придать точный теоретико-множественный смысл этим и некоторым другим терминам. Для записи спецификаций будем использовать язык формальных спецификаций RSL [6].

2 ОПРЕДЕЛЕНИЯ

Начнем с определения термина типа. За основу возьмем определение Бертрана Мейера, данное термину 'абстрактный тип данных' (АТД) [3].

Спецификация АТД состоит из четырех разделов:

- типа;
- сигнатура функций;
- аксиомы, описывающие типы;
- аксиомы, описывающие функции, в том числе и аксиомы-предусловия частично определенных функций.

Далее, сама спецификация будет называться типом. Доменом $Dom(X)$ типа X будет называться множество всех элементов, удовлетворяющих спецификации.

При описании примера на языке формальных спецификаций RSL [6] , в разделе типа (*type*) определим обозначения домена типа $Figure$ и дополнительный домен $UReal$ - это множество всех рациональных чисел, больших нуля.

В разделе величин (*value*) определим сигнатуры функций. Из всего набора функций выделим функции-генераторы (generator, на выходе функции есть обозначение домена типа) и функции-наблюдатели (observer, на выходе функции нет обозначения домена типа). Особенно важную роль играют функции, не выводимые из других функций спецификации.

В разделе аксиом (*axiom*) - аксиомы, описывающие попарное применение невыводимых функций-генераторов и функций-наблюдателей.

> В простых и привычных случаях, когда это не приводит к путанице, домен типа можно называть типом. Например, можно называть типом прямое произведение двух множеств $A \times B$, подразумевая, что есть спецификация функций проекции. Обозначение $A \times B$, мыслимое без функций проекции, - это домен типа, $A \times B$, мыслимое вместе с аксиомами функций проекции, - это тип.

Далее, уточним следующее определение Мейера [2] :

```
Класс - абстрактный тип данных, поставляемый
с, возможно, частичной реализацией.
```

Раз класс - это реализация типа, то тип будем считать спецификацией класса. Текст спецификации, то есть, типа, в которой произведена замена аксиом, описывающих попарное поведение функций, на частично рекурсивные функции, будет называться аппликативной реализацией. Текст с классом на языке программирования L, который удовлетворяет требованиям спецификации, будет называться (обычно императивной) L - реализацией.

3 СПЕЦИФИКАЦИЯ ПРИМЕРА

Вопросы единственности типа, полноты типа (в смысле полноты системы аксиом), противоречивости типа (в смысле противоречивости системы аксиом), единственности реализации, оптимальности спецификации и реализаций, вопросы синтаксического сахара RSL и др. не рассматриваются.

Попытаемся описать типы упомянутых выше C++ реализаций *Rectangle* и *Square* на языке формальных спецификаций RSL согласно принятому в RDG методу [4]. Дополнительный домен $UReal$ (то есть, все вещественные значения, большие 0) вводится, чтобы функции были всюду определенные и не надо было заниматься предусловиями.

3.1 Тип *Rectangle*

Аксиомы Rgw_sw и Rgh_sh очевидные. Что положил, то и вернул. Аксиомы Rgw_sh и Rgh_sw означают независимость высоты от ширины в значениях из множества $Figure$ по спецификации *Rectangle*. Например, для любой фигуры $f : Figure$, значения, возвращаемые функцией $GetWidth$, не зависят от применения функции $SetHight$.

```
---- File:./rsl/rectangle.rsl


scheme Rectangle =  class
  type
    Figure,
    UReal  =  {| r: Real :- r > 0.0 |}
  value
    SetWidth  : Figure >< UReal -> Figure,
    SetHeight : Figure >< UReal -> Figure,
```

```
    GetWidth  : Figure          -> UReal,
    GetHeight : Figure          -> UReal
  axiom
    [Rgw_sw]
      all w: UReal, f: Figure:- GetWidth(SetWidth(f, w)) = w,
    [Rgw_sh]
      all h: UReal, f: Figure:- GetWidth(SetHeight(f, h)) = GetWidth(f),
    [Rgh_sh]
      all h: UReal, f: Figure:- GetHeight(SetHeight(f, h)) = h,
    [Rgh_sw]
      all w: UReal, f: Figure:- GetHeight(SetWidth(f, w)) = GetHeight(f)
end

---- End Of File:./rsl/rectangle.rsl
```

Аппликативная реализация типа *Rectangle* может иметь следующий вид:

```
---- File:./rsl/ar.rsl


scheme AR =  class
  type
    Figure = UReal >< UReal,   -- фигура - пара (высота, ширина)
    UReal  =  {| r: Real :- r > 0.0 |}
  value
    SetWidth  : Figure >< UReal -> Figure
    SetWidth ((h, w), v) is  (h, v),
    SetHeight : Figure >< UReal -> Figure
    SetHeight ((h, w), v) is  (v, w),
    GetWidth  : Figure        -> UReal
    GetWidth (h, w) is  w,
    GetHeight : Figure        -> UReal
    GetHeight (h, w) is  h
end

---- End Of File:./rsl/ar.rsl
```

В этой реализации доменом у типа *Rectangle* является множество $Figure$, которое объявляется прямым произведением $UReal \times UReal$, функции $GetHeight, GetWidth$ - проекции на соответствующие сомножители.

3.2 Тип *Square*

В *Square*, предположительно, подтипе *Rectangle* аксиомы *Sgw_sh* и *Sgh_sw* имеют совершенно другой смысл, чем аналогичные аксиомы в *Rectangle*. Эти аксиомы явно требуют, что значения, возвращаемые функцией *GetWidth*, зависят от применения функции *GetHeight*:

```
---- File:./rsl/square.rsl


scheme Square =  class
  type
    Figure,
    UReal  =  {| r: Real :- r > 0.0 |}
  value
    SetWidth  : Figure >< UReal -> Figure,
    SetHeight : Figure >< UReal -> Figure,
    GetWidth  : Figure        -> UReal,
    GetHeight : Figure        -> UReal

  axiom
    [Sgw_sw]
      all w: UReal, f: Figure:- GetWidth(SetWidth(f, w))= w,
    [Sgw_sh]
      all h: UReal, f: Figure:- GetWidth(SetHeight(f, h)) = h,
    [Sgh_sh]
      all h: UReal, f: Figure:- GetHeight(SetHeight(f, h)) = h,
    [Sgh_sw]
      all w: UReal, f: Figure:- GetHeight(SetWidth(f, w)) = w,
end

---- End Of File:./rsl/square.rsl
```

Например, функция *GetWidth* возвращает то, что было передано в функцию *SetHeight*.

Аппликативная реализаци типа *Square*, записанная в форме наследования:

```
---- File:./rsl/as.rsl


AR
scheme AS = extend  hide  SetWidth, SetHeight in AR  with  class
  value
    SetWidth  : Figure >< UReal -> Figure
```

```
    SetWidth ((h, w), v) is  (v, v)
    ,
    SetHeight : Figure >< UReal -> Figure
    SetHeight ((h, w), v) is  (v, v)
end

---- End Of File:./rsl/as.rsl
```

Новый класс AS 'наследует' (*extend*) все из класса AR, при этом 'прячет' (*hide*) родительские функции $SetWidth$, $SetHeight$ и объявляет свои.

4 АППЛИКАТИВНАЯ РЕАЛИЗАЦИЯ ПРИМЕРА

Запишем аппликативную реализацию функции $LSPV$ из примера 1 на RSL. Текст этой функции мог бы выглядеть так:

```
---- File:./rsl/v.rsl


AS
scheme V = extend  AS  with  class
  value
    LSPV : Figure >< (Figure >< UReal -> Figure)  -> Unit
    LSPV ( r, setW) is
    let  h = 4.0,
          w = GetWidth (SetHeight( setW(r,5.0),h))   -- w = 4.0
       in
       if ( h * w = 20.0)  then skip else chaos end
    end
    pre
      (all h: UReal :- GetWidth(SetHeight(r, h)) = GetWidth(r)) /\  --  [Rgw_sh]
      (all w: UReal :- GetWidth(setW(r, w))      = w  )             --  [Rgw_sw]
end

---- End Of File:./rsl/v.rsl
```

Аппликативная реализация функции $LSPV$ делает точно то же, что и C++ реализация. Эффект позднего связывания удается достичь тем, что в $LSPV$ вместе с величиной типа $Figure$ передается функция $SetWidth$ из схемы $Square$. Это второй параметр функции $LSPV$, который имеет функциональный тип $Figure \times UReal \rightarrow Ureal$.

```
---- File:./rsl/uv.rsl


V
scheme UV = extend  V  with  class
    value
      rec : Figure

    test_case
     [usage]
       LSPV (rec, SetWidth)
  end

---- End Of File:./rsl/uv.rsl
```

Функция $LSPV$ последовательно применяет функции $setW$ и $SetHeight$ к полученному на вход значению $r : Figure$ и, в случае, если произведения высоты r на длину r не совпадает с 20, вызывает никогда незавершающееся выражение ($chaos$). Для того, чтобы подчеркнуть наличие ошибки к функции, всюду определенной согласно сигнатуре, приписано предусловие, означающее что функция написана в предположении аксиом спецификации *Rectangle* (а именно [Rgw$_s h$][$Rgw_s w$]).

Предусловие утверждает, что программист ожидает независимость координат (высоты и ширины) друг от друга:

```
 pre
   (all h: UReal :- GetWidth(SetHeight(r, h)) = GetWidth(r))  /\ --  [Rgw_sh]
   (all w: UReal :- GetWidth(setW(r, w))      = w )              --  [Rgw_sw]
```

На деле, поскольку в функцию $LSPV$ была передана функция $SetWidth$ из схемы AS - реализации $Square$ (аналогичная функция из схемы AR - реализации $Rectangle$ была скрыта), значение $w * h$ будет равно 16. То есть приводит к попаданию $LSPV$ на случай $chaos$, что означает нарушение спецификации. Тогда, в данном контексте, выражение 'изменение поведения функции' можно трактовать как нарушение спецификации. Ожидалось, что $LSPV$ это всюду определенная функция, но для функции $SetWidth$ из схемы AS - она 'изменила свое поведение' и её вычисление перестало завершаться.

Строго говоря, аргументация Мартина, почему его пример является настоящей проблемой, выглядит не слишком убедительно:

```
So here is the real problem: Was the programmer who wrote
that function justified in assuming that changing the
width of a Rectangle leaves its height unchanged? Clearly,
the programmer of LSPV made this very reasonable assumption.

Теперь у нас настоящая проблема: должен ли программист,
написавший функцию, быть осуждаем в его предположении что
изменение ширины Rectangle не изменяет его высоту? Ясно, что
программист функции LSPV сделал  осмысленное предположение.
```

То есть Мартин считает, что программист осуждаем быть не должен. Совершенно очевидно, что нет никакой проблемы и что программист должен делать только те предположения, которые записаны в спецификации и не делать никаких предположений от себя. Кроме того, (если рассуждать уже совсем формально) без наличия спецификации можно утверждать, что рассмотренная C++ реализация функция $LSPV$ при получении на вход переменной $a : Square$ сделала именно то, что требовалось - а именно аварийно завершила работу ($assert$). И именно этого и хотел программист. Поэтому нет никакой возможности говорить о каком-либо изменении поведения $LSPV$.

Вот если бы была написана хотя бы частичная спецификация этой функции, хотя бы было указано что $LSPV$ - всюду определенная функция ($LSPV : Rectangle \rightarrow Unit$), а при получении $a : Square$ она аварийно завершила работу, то есть оказалась бы не всюду определенной, то это была бы действительно проблема. Функция бы несоответствовала спецификации и можно было бы говорить об изменении ее поведения.

Если говорить неформально, то наличие аппликативной реализации и C++ реализации, приводящих к одинаковым трудностям, говорит о том, что корень проблемы находится не в языке реализации.

5 УТОЧНЕНИЕ ТЕРМИНОВ

Под термином выделение типа (subtyping) будем понимать добавление в спецификацию новых аксиом, не противоречащих аксиомам первоначальной. Исходная спецификация будет называться надтипом, спецификация, расширенная новыми аксиомами, будет называться подтипом.

То есть под выделением типа будем понимать уточнение спецификации, практически в соответствии с принципами проектирования по дизайну [1].

Посмотрим, что произойдет при попытке добавить к типу *Rectangle* аксиомы типа *Square*. Рассмотрим аксиомы Rgw_sh и Sgw_sh. Одну из *Rectangle*:

```
,[Rgw_sh]
 all h: UReal, f: Figure:-
          GetWidth(SetHeight(f, h)) = GetWidth(f)
```

Вторую из *Square*:

```
,[Sgw_sh]
 all h: UReal, f: Figure:- GetWidth(SetHeight(f, h)) = h
```

И запишем их вместе, соединив коньюнкцией, в виде следующей аксиомы:

```
all h: UReal, f: Figure:-
          h =   GetWidth(SetHeight(f, h))   /\
                 GetWidth(SetHeight(f, h)) = GetWidth(f)
```

через приравнивание $GetWidth(SetHeight(f, h))$ получаем

```
all h: UReal, f: Figure :-  h  = GetWidth(f)
```

Возьмем в $UReal$ значение w, причем $w \neq h$. Затем, раз f может быть любое, то, используя аксиому Rgw_sw, возьмем такое f, у которого ширина была задана равной w($f = SetWidth(f1, w)$). Получаем

```
    h =   GetWidth(f) = GetWidth(SetWidth(f1, w)) = w
```

Противоречие. Это означает, что две различные аксиомы Rgw_sh и Sgw_sh не могут выполняться вместе. Отсюда следует, что нет никакой возможности в принципе считать *Square* подтипом *Rectangle*. Можно считать их 'братьями', каждый из которых является подтипом некоторого другого надтипа.

Это же заключение можно сделать и из наблюдения, что пример можно было написать наоборот. Сначала написать C++ -реализацию *Square*, из которой позже пронаследовать C++ -реализацию *Rectangle*. Было бы совершенно странно иметь в языке возможность записать наследование реализации надтипа из реализации подтипа.

Пока, в свете предыдущих договоренностей, принцип подстановки Лисков может трактоваться следующим образом:

```
Пусть T и S некоторые классы. Если любая программа, использующая
обращения к переменной t: T, продолжает удовлетворять своей
спецификации при присвоении t значения  переменной s: S,
то тип класса S является подтипом  типа класса T.
```

6 ОТНОШЕНИЕ НАТУРАЛЬНЫХ И ЦЕЛЫХ

Попробуем посмотреть на типы - спецификации двух очень похожих классов: $UInt$ - реализация натуральных чисел и Int - реализация целых чисел, обладающих практически одинаковым набором операций. Затем проверим список их аксиом на совместимость как и в примере Мартина.

Спецификация множества натуральных чисел принадлежит Пеано, схема на RSL взята из [7, стр. 51]. Наличие или отсутствие нуля среди натуральных чисел не является принципиальным и является вопросом трактовки (см. [5, стр. 48]). Так же, с целью уменьшения громоздкости, удалены неиспользуемые аксиома порядка и операция умножения:

```
---- File:./rsl/peano.rsl


scheme PEANO =
class
  type
      N                     -- обозначение домена типа
  value
      zero : N,             -- завели величину zero  0
      succ: N -> N          -- функция следования succ(n) = n+1
  axiom
    [first_is_zero]         --      n+1 ~= 0
      all n : N :-          -- для каждого  n из N
       ~(succ(n) is zero), -- применение succ(n) не есть zero
   [induction]              -- аксиома индукции
      all p : N -> Bool:-   -- для любого предиката
              (p(zero) /\ (all n : N :- p(n) => p(succ(n)))) =>
                       (all n : N :- p(n))
end

---- End Of File:./rsl/peano.rsl
```

К аксиомам Пеано добавим операцию сложения.

```
---- File:./rsl/nat.rsl


PEANO       --           наследование схемы PEANO
scheme NAT = extend PEANO with
class
  value
```

```
    plus : N >< N -> N
  axiom
    [plus_zero]                -- n + 0 = n
      all n : N :-    plus(n, zero) is n,
    [plus_succ]
      all n1, n2 : N :-        -- n1 + (n2+1) = (n1 + n2)+1
      plus(n1, succ(n2)) is succ(plus(n1, n2))
end

---- End Of File:./rsl/nat.rsl
```

Чтобы получить спецификацию целых чисел, добавим функцию взятия обратного элемента и сопутствующие ей аксиомы:

```
---- File:./rsl/z.rsl


NAT
scheme Z = extend NAT with
class
  value
    minus : N -> N
  axiom
    [minus_zero]                  --  (-0) = 0
       minus(zero) = zero,
    [plus_minus]                  --  (-n) + n  = 0
      all n: N :-  plus(minus (n), n) is zero,
    [plus_minus_muinus]
      all n1, n2 : N :-          -- (-n1)+(-n2) = -(n1+n2)
      plus(minus(n1), minus(n2)) is minus(plus(n1, n2)),
    [minus_minus]                  --  -(-n) = n
      all n : N :-    minus(minus(n)) is n
end

---- End Of File:./rsl/z.rsl
```

Рассмотрим значение $succ(zero)$. Во-первых, по аксиоме $plus\_minus$

```
  plus(minus (succ(zero)), succ(zero)) is zero  --  (-1) + 1 = 0
```

Далее, по аксиоме $plus\_succ$ поменяем порядок вызова функций $succ$ и $plus$

```
  plus(minus (succ(zero)), succ(zero)) is       -- ((-1)+0) + 1 = 0
           succ(plus (minus (succ(zero)), zero ))
```

Осталось по $plus\_zero$ удалить сумму и получить утверждение, противоречащее аксиоме $first\_is\_zero$:

```
succ(plus (minus (succ(zero)), zero ))  is  -- (-1) + 1 = 0
     succ(minus (succ(zero)))
```

$minus(succ(zero))$ является таким n из N, что применение к нему функции $succ$ дает $zero$. Противоречие и явное несоответствие принципу подстановки-замещения. Раз любая работающая программа, написанная в терминах класса $UInt$, (а значит и такая, которая написана в предположении $first\_is\_zero$) должна продолжать работать, если ей на вход поставляется любой обьект из класса Int (а значит и такой, для которого аксиома $first\_is\_zero$ выполняться не будет. Существование таких объектов гарантируется согласно спецификации типа целых чисел). Это означает что ни тип Z не может быть подтипом типа Nat, ни тип Nat не может быть подтипом типа Z.

Рассмотрим подробнее, что происходит с доменами типа в наших примерах. В рассмотренных реализациях домен $Figure$, удовлетворяющий типу $Square$, это множество пар одинаковых неотрицательных рациональных чисел (обозначим $Figure_{Square}$), а домен $Figure$, удовлетворяющий $Rectangle$, это множество всех пар неотрицательных рациональных рациональных (обозначим $Figure_{Rectangle}$). То есть, $Figure_{Square}$ является подмножеством $Figure_{Rectangle}$. В случае натуральных и целых N_{Nat} является подмножеством N_Z. Кроме того, заметим, что в обоих примерах, наследование можно записывать как $Rectangle$ от $Square$ (а Int от $UInt$), так и наоборот; Получаем, что при построении из некоторого надтипа A подтипа B можно получить три случая:

- пока неинтересный случай, когда домен типа остается неизменным.
- домен подтипа 'увеличивается': $Dom(A) \subset Dom(B)$;
- домен подтипа 'уменьшается': $Dom(A) \supset Dom(B)$.

По аналогии с примером Мартина рассмотрим случай уменьшения домена на примере натуральных целых. Это значит, что сначала написали спецификацию Z, в предположении что функция $succ(minus(succ(zero)))$ дает zero. Потом, в соответствии со спецификацией написали пример $LSPV3$:

```
---- File:./rsl/v3.rsl
```

```
Z
scheme V3 = extend  Z  with  class
  value
    LSPV3 : N ><  (N -> N)  -> Unit
    LSPV3 ( a,   suc) is
      let one =  suc(zero)
      in
        if a = one /\ suc (minus(a)) ~= zero then chaos else skip end
      end
    pre
      suc(minus(suc(zero))) =  zero,
   LSPVM : N   -> N
   LSPVM ( a) is  minus(a)
end

---- End Of File:./rsl/v3.rsl
```

Потом попробовали получить как подтип спецификацию Nat, добавляя в нее аксиому $zero\_is\_first$. Потом его реализацию $UInt$, в которой

- *succ* никогда не возвращает *zero*;
- отсутствует реализация $minus$.

После чего оказывается, что либо функция $LSPVM$ испортит значения подкласса (нарушит инвариант), либо функция $LSPV3$ не сможет завершить свою работу, получив в качестве фактических параметров $succ(zero)$ и $UInt :: succ$. Создается ситуация, в точности аналогичная ранее рассмотренной с $LSPV$.

7 ПЕРЕСМОТР ПРОБЛЕМЫ

Теперь можно вспомнить, что было первой трудностью в разборе примера Мартина. Ею оказалось то, что объект $s : Square$, рассматриваемый как принадлежащий типу надкласса, после выполнения метода надкласса нельзя рассматривать как принадлежащий типу подкласса. Ибо после вызова методов надкласса (вследсвие раннего связывания обьекта и методов класса) оказывалось нарушено условие равенства сторон объекта $Square$ (инвариант $Square$). Но необходимость сохранять инвариант s есть, только при желании присвоения значения переменной типа $Rectangle$ переменной типа $Square$. Именно это влечет обязательность выполнения инварианта $Square$.

То есть, главное, что ожидается от пары надкласс и подкласс - это возможность точного (без преобразования значения) прямого и обратного присвоения значения по цепочке подкласс - надклас - применение метода - подкласс. Слово 'точное' важно. К примеру, на практике регулярно выполняется неточное преобразование значений из целых в рациональные и обратно.

Пусть есть класс A, в котором обьявлена функция $f : Dom(A) \times X \to Dom(A)$ (далее $A :: f$), и его наследник - подкласс B с функцией $f : Dom(B) \times X \to Dom(B)$ (далее $B :: f$), для некоторого множества X. Причем, $Dom(A) \neq Dom(B)$. Будем считать, что соответствующая функция, заданная на большем множестве, вообще говоря, может вырабатывать значения не попадающие в меньшее множество. Пусть объявлены переменная a из класса A, и переменная b из класса B. Рассмотрим, что может происходить при выполнении последовательности операторов:

$a = b$;

$a.f(x)$;

$b = a$;

Всего получаем четыре ситуации:

- В случае увеличения домена ($Dom(A) \subseteq Dom(B)$) точное начальное присвоение $a = b$ представляется проблематичным. Не каждое значение переменной b может быть присвоено переменной a. В таком случае оказывается невозможным использовать функции надкласса и все равно требуется выполнение всех аксиом надтипа;

- Случай уменьшения домена ($Dom(A) \supseteq Dom(B)$) и раннего связывания. То есть, при обращении к f - $a.f(x)$ используется функция $A :: f(x)$. Можно свободно присваивать значение $a = b$. Но для некоторого x получаем $a.A :: f(x)$ не принадлежит $Dom(B)$, что делает невозможным обратное присвоение значения $b = a$.

  По-видимому, это достаточное, но не необходимое условие. В случае, если $Dom(B)$ является чем-то вроде идеала во множестве $Dom(A)$, когда для функции $A :: g : Dom(A) \times Dom(A) \to Dom(A)$ и для любого b из $Dom(B)$ и любого a из $Dom(A)$ применение $A :: g(b, a)$ будет принадлежать $Dom(b)$, оказывается возможным обратное присвоение.

- Случай уменьшения домена и позднего связывания. То есть, при обращении к f - $a.f(x)$ используется функция $B :: f(x)$. Можно свободно присвоить значение $a = b$. Для любых значений x, $a.B :: f(x)$ принадлежит $Dom(B)$, возможно обратное присвоение $b = a$. Вопрос состоит в том, чтобы функция $B :: f$ удовлетворяла всем аксиомам типа A;

- в случае, когда домен типа не меняется, можно использовать и родительскую функцию $A :: f$ и дочернюю функцию $B :: f$, если она удовлетворяет аксиомам типа A;

8 ЗАКЛЮЧЕНИЕ

Осталось сформулировать следующее наблюдение:

во-первых, есть два текста - некоторая формальная математическая система, в которой записываются типы - спецификации, и язык программирования L, в котором записывается программа;

во-вторых, есть четыре операции:

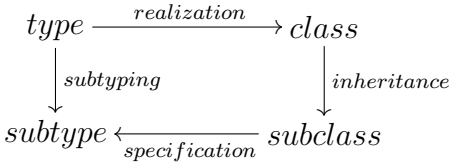
- выделение подтипа - subtyping;

- наследование (образование подкласса) - inheritance;

- L-реализация спецификации (записывание программы на языке L) - realization;

- восстановление спецификации по программе - specification.

Принцип подстановки Лисков показывает, при каких условиях операции 'реализация', 'образование класса', 'восстановление спецификации' будут приводить к тем же результатам, что и операция 'выделение подтипа', то есть, окажутся в некотором смысле коммутативными:



выделение подтипов и наследование

8.1 Настоящий надтип классов *Square* и *Rectangle*

В заключение в качестве иллюстрации запишем не полностью определенный тип - настоящий родитель обоих классов с удаленными аксиомами Rgw_sh, Rgh_sw или Sgw_sh, Sgh_sw:

```
---- File:./rsl/realdad.rsl

scheme RealDad =  class
    type
      Figure,
      UReal  =  {| r: Real :- r > 0.0 |}
    value
      SetWidth  : Figure >< UReal -> Figure,
      SetHeight : Figure >< UReal -> Figure,
      GetWidth  : Figure          -> UReal,
      GetHeight : Figure          -> UReal
    axiom
      [gw_sw]
        all w: UReal, f: Figure:- GetWidth(SetWidth(f, w)) = w,
      [gh_sh]
        all h: UReal, f: Figure:- GetHeight(SetHeight(f, h)) = h
  end

---- End Of File:./rsl/realdad.rsl
```

C++ -реализация которого может выглядеть следующим образом:

```
---- File:./rsl/realdad.cpp

class RealDad{
  public:
    virtual void SetWidth(double w)=0;
    virtual void SetHeight(double h)=0;
    double GetHeight() const {return itsHeight;}
    double GetWidth()  const {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};

---- End Of File:./rsl/realdad.cpp
```

Для сохранения свойства быть подтипом, оба класса *Square* и *Rectangle* должны наследоваться от *RealDad*, который и должен использоваться для написания полиморфных функций вроде $LSPV$ в качестве класса формального параметра.

Предметный указатель

ССЫЛКИ

[1] Бертран Мейер. Основы объектно-ориентированного программирования. Техника наследования. http://www.intuit.ru/department/se/oopbases/16/11.html.

[2] Бертран Мейер. Основы объектно-ориентированного программирования. Статические структуры: классы. http://www.intuit.ru/department/se/oopbases/7/.

[3] Бертран Мейер. Основы объектно-ориентированного программирования. Абстрактные типы данных (АТД). http://www.intuit.ru/department/se/oopbases/6/10.html.

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

[5] И.В.Арнольд. Теоретическая арифметика, 1938. Учпедгиз, М, 1938, http://agp.hx0.ru/arts/teor_arifm.djvu.

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

[7] The RAISE Method Group. The RAISE SPECIFICATION LANGUAGE, 1992. Prentice Hall Europe, Denmark, 1992.

[8] Robert C.Martin. The Liskov Substitution Principle, 2003. C++ Report, March 1996, http://www.objectmentor.com/resources/articles/lsp.pdf.