УДК 681.3.06

Пискунов А.Г., к.ф-м.н., с.н.с

A.G. Piskunov, Ph.D

Об отличиях между понятиями типа и класса

About the differences between the notions of type and class

Национальный авиационный университет
г. Киев, 03165, просп. Космонавта Комарова, 1,
e-mail: agp1.ua@gmail.com

National Aviation University of Kyiv,
03165, Kyiv, 1,Kosmonavta Komarova str.,
e-mail: agp1.ua@gmail.com

У статті застосовується метод розробки програмного забезпечення RAISE від групи авторів RAISE Development Group (далі RDG) для формального аналізу приклада Роберта Мартіна, що ілюструє принцип підстановки (або інакше принцип заміщення) Барбари Лісков. Таким способом вдалося уточнити поняття типу, успадкування, виділення підтипу і взаємозв'язок між ними.

Під типом пропонується розуміти набір вимог (або аксіом) A до деякої множини значень X і до деякої множини функцій F, заданих на множині X, тобто трійку T = {A, X, F}. Виділенням підтипу назвемо додавання до набору A нових вимог і/або нових функцій до набору F. Зрозуміло, що нові вимоги не повинні суперечити вже існуючим. Такі зміни в аксіомах і функціях можуть спричинити необхідність заміни початкової множини значень X на деяку іншу – X1. Таким чином, отримуємо нову трійку T1 = {A1, X1, F1}, яка буде називатися підтипом типу T. Класом, слідом за Бертраном Мейєром будемо називати реалізацію типу в деякій мові програмування.

Вдалося показати, що тип класу спадкоємця, не є підтипом батьківського класу, так як набори аксіом з типів спадкоємця і батька виявляються несумісними.

Ключові слова: тип, клас, RAISE Development Method.

In the article, the method of software development RAISE, after a group of authors called the RAISE Development Group (RDG further) is using for formal analysis of some example by Robert Martin. This example illustrates the Liskov substitution principle. In this way it was possible to clarify the concepts of type, inheritance, subtyping and the relation between all of them. Let the triplet T = (A, X, F), where A is the set of requirements (or axioms) A to some set of values of X and to some set of functions F defined on the set X is called the type. If some new requirements would be added to A or/and some new functions would be added to F then the new triplet T1 = (A1, X1, F1) would be called the subtyping. It is clear that the new requirements should not contradict the existing ones. There is a new set of values of X1 in triplet T1 because of discussed modification of the set of axioms A1 and/or the set of functions F1. And T1 will be called subtype of T. Class, followed by Bertrand Meyer, will be called any implementation of a type in any programming language. It was shown that the class inheritance can lead to the violation of the property to be a subtype. That is, the type of successor class, generally speaking, is not a subtype of the type the parent class. It was happend because the sets of axioms of the successor type and parent type are incompatible. ***Some- certain***

Keywords: type, class, RAISE Development Method.

Статтю представив професор Буй Д.Б.

1. Введение

Понятию *тип* уделяли внимание многие авторы, например, Деннис Бьорнер (Dines Bj0rner) [1] или Люка Карделли [2]. Но эти их работы носили описательный характер, что ограничивает возможность использования в короткой статье. А принцип подстановки Лисков короткий и, поэтому, может быть подвергнут формальному анализу. В работе будет рассмотрен пример из статьи Роберта Мартина [3], показывающий некоторое несоответствие между наследованием в языке C++ и выделением подтипа.

Принцип подстановки Лисков Мартин формулирует 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: «если для каждого объекта *o1* типа *S* существует объект *o2* типа *T* такие что, любая программа P, записанная в терминах *T*, не изменяет своего поведения при подстановке объекта *o1* вместо объекта *o2*, то тип *S* является подтипом *T*». Для лучшего понимания принципа, 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 приведенном 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 возражение 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* [5].

2. Определения

Начнем с определения термина типа. За основу возьмем определение Бертрана Мейера, данное термину *абстрактный тип данных* (АТД) [6, стр.113]. Спецификация АТД состоит из четырех разделов: типы; сигнатуры 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* [5], 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 [6, стр.122]: класс – абстрактный тип данных, снабженный некоторой, возможно частичной, реализацией. Раз класс – это реализация типа, то тип естественно 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. Две другие аксиомы *Rgw_sh* и *Rgh_sw* означают взаимную 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* × *UReal*, функции *GetHeight*, *GetWidth* – проекции на соответствующие сомножители.

3.2. Тип *Square*

В *Square*, предположительно, подтипе *Rectangle* аксиомы *Rgw_sw* и *Rgh_sh* имеют тот же смысл, что и ранее, а смысл аксиом *Sgw_sh* и *Sgh_sw* совершенно поменялся. Эти две последние аксиомы явно утверждают, что значения, возвращаемые функцией *GetWidth*, зависят от применения функции *SetHeigh u,*

наоборот, GetHeight возвращает результат предыдущего использования *SetWidth*:

```
---- 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 реализация типа *Square*, записана 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 Мартина (см. раздел *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* × *UReal* → *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 произведение высоты *r* на длину *r* не совпадает с числом 20, вызывает никогда не завершающееся 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_sh* и *Rgw_sw*).

Предусловие утверждает, что программист ожидает независимости 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 функции *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 → Unit*), а при получении a : *Square* она аварийно завершила работу, то это была бы действительно проблема. Так как домен типа Square входит в домен типа Rectangle, то оказывается что функция LSPV не является полностью определенной, что влечет несоответствие спецификации.

Далее, 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 первоначальным аксиомам.

Исходная спецификация будет называться надтипом, спецификация, расширенная новыми аксиомами, будет называться подтипом. То есть, под выделением типа будем понимать уточнение спецификации.

Посмотрим, что произойдет при попытке добавить к типу *Rectangle* аксиомы типа *Square*. Рассмотрим аксиомы *Rgw_sh u 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)
```

в связи с общей частью *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.*

Имеем 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. [8, 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
succ(plus (minus (succ(zero)), zero))
-- ((-1)+0) + 1 = 0.

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. 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*. Например 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 (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 *LSPV 3*:

```
---- 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
```

Далее 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 (вследствие раннего связывания объекта 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, одно из множеств строго включается во второе. 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 $b = a$.

  Примечание. *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 (рис. 1).

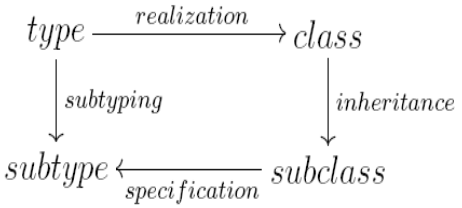

Рис. 1. Выделение подтипов и наследование

8.1. Настоящий надтип классов *Square* и *Rectangle*

Запишем не полностью определенный тип – настоящий родитель обоих классов с удаленными аксиомами *Rgw_sh*, *Rgh_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. Собственно, саму программу LSPV предъявлять в качестве контрпримера будет нельзя, так как в связи с удаленными аксиомами нельзя будет предполагать, что произведение извлеченной длины и ширины даст 20.

Список использованных источников

1. *Dines Bj0rner*. Software Engineering 1: Abstraction and Modelling / Dines Bj0rner /. [ISBN 3-540-21149-7]. – Springer, 2005.– 714 p.
2. *Luca Cardelli*, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism./ Luca Cardelli, Peter Wegner // Computing Surveys. – December 1985. – Vol. 17. – № 4. – P. 471–522.
3. *Robert C.Martin*. The Liskov Substitution Principle / Robert C.Martin. // C++ Report. – March 1996.
4. *The RAISE Method Group*. The RAISE Development Method // BCS Practitioner Series [ISBN 0-13-752700-4]. – Prentice Hall. –1995.
5. *Chris George*. Introduction to RAISE./ Chris George. – UNU-IIST report. – 2002. – No. 249.
6. *Bertrand Meyer*. Object-Oriented Software Construction / Bertrand Meyer. – Prentice Hall. [ISBN 0-13-629155-4]. – 1997.
7. *The RAISE Method Group*. The RAISE SPECIFICATION LANGUAGE. – Prentice Hall Europe, Denmark. – 1992.
8. *Арнольд И. В*. Теоретическая арифметика / И. В.Арнольд. – М.: Учпедгиз, 1938. –399 с.

References

1. DINES BJ0RNER. (2005) *Software Engineering 1: Abstraction and Modelling (Texts in Theoretical Computer Science. An EATCS Series)*. Springer. ISBN 3-540-21149-7.
2. LUCA CARDELLI, PETER WEGNER. (1985) *On Understanding Types, Data Abstraction, and Polymorphism*. Computing Surveys, December 198, Vol 17 n. 4, pp 471-522.
3. ROBERT C.MARTIN. (1996) *The Liskov Substitution Principle*. C++ Report, March 1996.
4. The RAISE Method Group. (1995) *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall. ISBN 0-13-752700-4.
5. CHRIS GEORGE. (2002) *Introduction to RAISE.* UNU-IIST report No.249.
6. BERTRAND MEYER. (1997) *Object-Oriented Software Construction*. Prentice Hall, ISBN 0-13-629155-4.
7. The RAISE Method Group. (1992) *The RAISE SPECIFICATION LANGUAGE*. Prentice Hall Europe, Denmark.
8. ARNOLD, I.V.. (1938) *Theoretical arithmetic*. Moskva.

Надійшла до редколегії 15.09.15