УДК 519.681.2
DOI https://doi.org/10.32689/maup.it.2023.5.7

Олексій ПІСКУНОВ
кандидат фізико-математичних наук, старший науковий співробітник, доцент кафедри прикладної математики, Національний авіаційний університет, просп. Любомира Гузара, 1, Київ, Україна, індекс 03058 (oleksii.piskunov@npp.nau.edu.ua)

ORCID: 0000-0002-9200-3422

Наталя ТУПКО
кандидат фізико-математичних наук, доцент, доцент кафедри прикладної математики, Національний авіаційний університет, просп. Любомира Гузара, 1, Київ, Україна, індекс 03058 (natalia.tupko@npp.nau.edu.ua)

ORCID: 0000-0003-0625-3271

Іван ПЕТРЕНКО
інженеренер-програміст, Splunk, вулиця Добрий пастир, 122б, кв. 37, Краків, Польща, індекс 31-416 (humty.ua@gmail.com)

ORCID: 0009-0000-3409-0022

Oleksii PISKUNOV
Candidate of Physical and Mathematical Sciences, Senior Research Fellow, Associate Professor at the Department of Applied Mathematics, National Aviation University, 1, Lubomyra Huzara Ave, Kyiv, Ukraine, postal code 03058 (oleksii.piskunov@npp.nau.edu.ua)

Natalia TUPKO
Candidate of Physical and Mathematical Sciences, Associate Professor, Associate Professor at the Department of Applied Mathematics, National Aviation University, 1, Lubomyra Huzara Ave, Kyiv, Ukraine, postal code 03058 (natalia.tupko@npp. nau.edu.ua)

Ivan PETRENKO
Software Engineer, Splunk, 122b, app. 37, Dobryi pastyr St, Krakow, Poland, postal code 31-416 (humty.ua@gmail.com)

Бібліографічний опис статті: Піскунов, О., Тупко, Н., Петренко, І. (2023). Алгебраїчне проєктування програмного забезпечення. *Інформаційні технології та суспільство*, 5 (11), 50–59. DOI: https://doi.org/10.32689/maup.it.2023.5.7

Bibliographic description of the article: Piskunov, O., Tupko, N., Petrenko, I. (2023). Alhebraichne proiektuvannia prohramnoho zabezpechennia [Algebraic software design]. *Informatsiini tekhnolohii ta suspilstvo – Information technology and society,* 5 (11), 50–59. DOI: https://doi.org/10.32689/maup.it.2023.5.7

АЛГЕБРАЇЧНЕ ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Анотація*. У статті розглянуто алгебраїчний підхід до проєктування та тестування програмного забезпечення.* ***Метою*** *статті є розробка класу, який реалізує арифметику Пеана. Арифметика Пеана є однією з базових конструкцій при аксіоматичному побудуванні математики. Клас арифметики, який представлений в статті є півгрупою натуральних чисел і є першим в ієрархії числових класів (група цілих чисел, кільце цілих чисел, поле раціональних чисел) для демонстрування небезпечності універсального поліморфізму підтипів та порушення принципу підстановки Лісков.* ***Методи дослідження:*** *під час дослідження використовуються базові положення методу проєктування по контракту Бертрана Меєра та методу формальної розробки RAISE, які дозволяють застосовувати формальну логіку.* ***Наукова новина дослідження*** *полягає в тому, що змінене трактування до функціонального типу методів класу та більш широке використання вимог до класу у вигляді алгебраїчних рівностей, якими описуються вимоги до інваріантів, передумов та післяумов. Об'єднання вимог до функціональних типів і цих рівностей логічною зв'язкою 'і' дозволяє однозначно прогнозувати порушення принципу підстановки та вимагає змінення правила підстановки (subsumption). Крім цього, звертається увага на властивість категоричності відповідної алгебраїчної моделі, яка робить неможливими некоректні реалізації, на прямі аналогії між аксіоматичним викладенням математичної теорії та розробкою технічного завдання програмістам. Також, даний підхід значно спрощує підбір тестів для димового тестування (smoke testing) програмного забезпечення.* ***Висновки.*** *Алгебраїчне проєктування та тестування базується на математичних принципах, що дозволяє: уникати двозначності і неоднозначності в описі функціональності; забезпечувати точність та однозначність у формулюванні вимог до програми; автоматизувати процес генерації тестових випадків та перевірки роботи, відповідно підвищувати надійність; виявляти*

і усувати помилки ще на стадії розробки та прискорювати розробку програмного забезпечення. Запропоновані в статті доповнення до алгебраїчного підходу проєктування по контракту та методу формальної розробки RAISE демонструють універсальність алгебраїчного проєктування, яке дозволяє перейти від мистецтва програмування та мистецтва тестування програм до формальних технологічних прийомів.

Ключові слова: *алгебраїчне проєктування, алгебраїчне тестування, метод RAISE, димове тестування, проєктування за контрактом, принцип підстановки, небезпечність універсального поліморфізму підтипів.*

ALGEBRAIC SOFTWARE DESIGN

Abstract. *The article discusses an algebraic approach to designing and testing software. The purpose of the article is to develop a class that implements Peano arithmetic. Peano arithmetic is one of the fundamental constructs in axiomatic mathematics. The arithmetic class presented in the article represents a semigroup of natural numbers and this class is the first example in the hierarchy of numerical classes (integers, integer rings, rational number fields) that demonstrates the potential unsafety of universal inclusion polymorphism and violations of the Liskov substitution principle.* ***Research methods.*** *During the study, basic principles of the Bertrand Meyer's design by contract method and the formal development method RAISE are used, which allow applying formal logic.* ***Scientific novelty.*** *The modified interpretation of the functional type of class methods and the consistent use of requirements in the form of algebraic equalities make it possible at the time of design to indicate the unsafety of universal inclusion polymorphism. Additionally, attention is drawn to the categoricality (rigitity) of algebraic model, which makes incorrect implementations impossible, and to direct analogies between the axiomatic presentation of mathematical theory and the development of specifications. Furthermore, this approach significantly simplifies the of smoke test design.* ***Conclusions.*** *Algebraic design and testing are based on mathematical principles, allowing for the avoidance of ambiguity and uncertainty in functionality descriptions, ensuring accuracy and unambiguity in formulating specifications, automating the process of test cases design and verification of software requirements, thereby makes it easier to detect and correct a design and coding errors.*

Key words: *unsafety of universal inclusion polymorphism, functional type of class method, design by contract, RAISE method, algebraic equalities.*

Постановка проблеми та аналіз останніх досліджень і публікацій. Поліморфізм підтипів (шаблонів) згідно з правилом підстановки, який використовується в класичних мовах програмування, наприклад C++ та C#, дозволяє передавати змінні похідного класу, тобто наступного, у функції, які написані в термінах базового класу [7]. В загальному розумінні, ця властивість постулюється як основна перевага мов об'єктно-орієнтованого програмування (ООП) над мовами, які не належать до ООП. Стверджується, що при цьому можна розробляти алгоритми, які однаково успішно працюють як зі змінними базового класу, так і зі змінними похідного. Відповідно у розроблені за такими алгоритмами поліморфній функції можна безпечно передавати змінні довільних похідних класів. При цьому досить давно в роботах Б. Лісков [21] було зауважено, що таке використання поліморфної функції є безпечним, якщо тип похідної функції збігається з типом базового класу. Один із широко відомих прикладів, який згадується в роботах Б. Мартіна з квадратами та прямокутниками [14], підтверджує спостереження Б. Лісков. Наведені факти заперечують думку розробників компіляторів класичних ООП мов, що успадкування являє собою формування підтипу та безпосередньо вказує на наявність проблеми. При цьому не пояснюється наперед як потрібно проєктувати базовий та похідний клас, щоб і успадкування, і використання поліморфних функцій було безпечним. Відповідно не зрозуміло, в яких випадках порушується принцип підстановки Б. Лісков, а в яких – ні. До того ж, автори прикладів (Р. Мартін) не замислюються про обговорення такої властивості як категоричність. Наприклад, у монографіях К. Дейт та Б. Меєр [10; 15] успадковують квадрати від прямокутників. З вище сказаного, напрошуються висновок: інструментальні засоби розробки без попереджень дозволяють використовувати небезпечний поліморфізм підтипів, який приводить до неочікуваного порушення специфікацій і виникає питання: яким чином правильно проєктувати потрібні класи?

Виклад основного матеріалу. Перехід до формального обговорення проблеми вимагає формальних визначень для понять, що використовуються. Згідно з джерелами [1; 2], термін «тип» буде вважатись синонімом до терміну «абстрактний тип даних». Термін «клас», за Б. Меєром [15], трактуватиметься як «запрограмований (реалізований) абстрактний тип даних». В якості властивості, яка має виконуватись для змінної базового класу [21], щоб формально дотримуватись принципу підставки, буде обрано наступну властивість: алгоритм проєктованої поліморфної функції задовольняє вимогам, що пред'являються до нього, тобто задовольняє своїй специфікації.

З цієї формальної точки зору, на основі загальних положень проєктування за контрактом Б. Меєра та формального методу розробки RAISE [12] приклад Р. Мартіна було розглянуто в роботі [2]. У ході зворотної розробки для класів прямокутників та квадратів були представлені відповідні їм абстрактні типи даних, а саме, як пояснюється в роботі [15], введена невизначена множина даних Figure та множина невід'ємних дійсних чисел UReal, для них задані функції «взяти довжину однієї сторони» (getWifth: Figure → UReal), «задати довжину другої сторони» (setHeight: Figure × UReal → Figure). При цьому, властивості цих функцій (і всіх інших необхідних у прикладі) задавалися у вигляді алгебраїчних рівностей:

$$\forall r : \mathrm{UReal}, f : \mathrm{Figure} \cdot \mathrm{getWidth}\big(\mathrm{setHeight}(f, r)\big) = \mathrm{getWidht}(f)$$

Тобто, для довільної фігури виконується вимога: яку б довжину однієї сторони не задавали, довжина другої сторони не змінюється. Отже, вимоги до значень параметрів функцій, до властивостей значень, які повертаються, та самих функцій визначаються за допомогою алгебраїчних рівнянь. Список усіх множин, сигнатур функцій та алгебраїчних рівностей утворював у термінах Б. Меєра [15] «контракт». Цей контракт повинен виконуватись при спадкуванні. Але в розглянутому прикладі це виявилося не так, контракт виявився порушеним. В об'єктах одного класу при зміні довжини однієї сторони змінювалася довжина іншої. В об'єктах другого класу довжина другої сторони не змінювалась. Зрозуміло, що це призводило до порушень роботи поліморфних функцій. Тобто методи, які змінюють вміст свого об'єкту, виявлялися причиною порушення роботи поліморфної функції.

Надалі, при розгляді цього прикладу довелося дещо змінити трактування поняття типу методів та наблизити його до типу функцій відповідного абстрактного класу. До списку класів формальних параметрів і класу значення, що повертається (які беруться з сигнатури методу) був доданий клас до якого належить метод. Наприклад, замість сигнатури методу f класу c (метод не змінює об'єкт this):

```
class c{
  int fv(T v);
}
```

Розглядається його функціональний тип з явним додаванням ще однієї множини допустимих значень для прихованого параметра this:

$$f : dom(c) \times dom(T) \to dom(int).$$

Зрозуміло, що у функціональному типі (методів, які приводять до порушення принципу підстановки) домен класу з'являється зліва і справа від функціональної стрілки.

Зауваження.

dom(c) – це множина допустимих кортежів, які можуть з'являтися в об'єктах класу c. Згідно з Л. Карделлі [8] (Objects as records), математична парадигма «об'єкти-як-записи» (тобто, як кортежі з прямих добутків деяких множин) цілком достатня для опису всіх основних властивостей об'єктів. Зважаючи на те, що функція є певним підмножиною прямого добутку області визначення та множини значень, будь-яка функція також може розглядатися як компонента такого кортежу. Отже, кортеж може містити числа, символи, рядки і, крім того, функції. Це впритул підводить поняття об'єкта до поняття абстрактного автомата. Що в свою чергу, означає: твердження Д. Парнаса [17]) «Змінна – це автомат» слід розуміти буквально. Спадкування абстрактних автоматів розглядалося в [18].

Далі буде потрібне визначення таких понять: метод, функція стану або переходу, конструктор, функція виходу.

Метод – це функція, яка визначена в області видимості класу. Функціональний тип будь-якого методу, крім статичного, повинен мати додатковий параметр порівняно з його сигнатурою ліворуч та/або праворуч від функціональної стрілки.

Функція переходу (у розумінні переходу з одного стану об'єкта в інший) – це функція, в сигнатурі якої домен класу з'являється ліворуч і праворуч від функціональної стрілки (→ для всюди визначеної функції або ↦ для частково визначеної функції). Згідно з [12] така функція називається генератором (generator).

Конструктор – це функція, в сигнатурі якої домен класу з'являється тільки праворуч від функціональної стрілки і відсутній ліворуч.

Функція виходу (функція – спостерігач (observer) – це функція, в сигнатурі якої домен класу з'являється лише ліворуч від функціональної стрілки і відсутній справа [12].

Аксіоматика Пеано та напівгрупа натуральних чисел

У цьому розділі до множини чисел, які визначаються аксіомами Пеано, додані аксіоми для функції односпрямованого ітератора: взяти наступне (successor) suc і функції додавання (summator) sum. Отже:

1. P_0 : існує число 1, яке не слідує ні за яким натуральним числом.
2. P_1 : кожному натуральному числу n однозначне відповідає безпосередньо наступне за ним натуральне число n'.
3. P_2 : будь-яке натуральне число n, за винятком 1, безпосередньо слідує за одним і тільки одним натуральним числом.
4. P_3 : Якщо твердження S доведено для 1 і якщо з припущення, що воно вірне для натурального числа n, випливає, що воно вірне для безпосередньо наступного натурального числа n', то твердження S виконується для всіх натуральних чисел.

Аксіома математичної індукції P_3 , вже реалізована в багатьох мовах програмування у вигляді ітерації (операторів циклу) та/або рекурсії, тому надалі не буде згадуватися. Дещо в іншому вигляді, з цією ж специфікацію на мові RSL можна ознайомитись в [22], розділ 'Example: The Natural Numbers'.

Множина натуральних чисел задовольняє перерахованим аксіомам і позначимо її через $\mathfrak{N}$. Специфікація для ітератора має вигляд:

$$P_4 : suc : \mathfrak{N} \to \mathfrak{N} \wedge \forall n \in \mathfrak{N} \cdot suc(n) = n'$$

Специфікація для суматора Грассмана [5] має вигляд:

$$P_5 : sum : \mathfrak{N} \times \mathfrak{N} \to \mathfrak{N} \wedge \forall a, r \in \mathfrak{N}:$$

$$sum(a,1) = suc(a) \wedge sum(a, suc(r)) = suc(sum(a,r))$$

Крім того, з двох останніх аксіом видно, що вони постулюють операцію порівняння двох чисел, тобто введення символу '='.

Дуже важливою характеристикою такої специфікації для функцій suc і sum виявляється наступне:

– неявність специфікації функцій, що є наслідком використання алгебраїчних рівностей [15] (ця неявність є ключовим аспектом лаконічного опису абстрактних типів даних та їх майбутніх аналогів – класів);

– аксіоми для відносин між функціями дають лаконічний опис властивостей специфікованого типу в суто математичних термінах без допомоги імперативних міркувань.

Якщо у роботах Б. Меєра процес вибору аксіоми для відносин між функціями згадуються не надто явно, то у авторів методу формальної розробки RAISE аксіоматичний опис попарного застосування функцій з'являється вже як обов'язкова вимога методу розробки:

– For each possible combination of non-derived observer and non-derived generator, define an axiom expressing the relation between them. We have three non-derived generators and two non-derived observers, so we have six such axioms. These axioms are called observer-generator axioms [12].

– Для кожної можливої пари складеної з функції – спостерігача (observer), яка не виводиться (тобто не може бути запрограмована за допомогою інших функцій класу) і функції- генератора (generator), яка не виводиться, визначити аксіому, що виражає відношення між цими функціями.

Самі розробники методу формальної розробки RAISE з питання алгебраїчного підходу до аксіоматичного опису відносин між функціями відсилають до робіт Johna Guttag [11], який, у свою чергу, відсилає до Хоара та Флойда: «The algebraic approach used here owes much to the work of Hoare (which in turn owes much to Floyd)».

Насправді, аксіоматичний опис відношення двох функцій, що специфікуються, ще раніше з'явився в алгебрі. Наприклад, саме так виглядає аксіома дистрибутивності $\mathbb{Z}_{VIII}$ в аксіоматиці кільця цілих чисел [20]:

$$\forall a,b,c \in \mathbb{Z} \cdot (a+b) \cdot c = a \cdot c + b \cdot c .$$

У функціональному записі ця аксіома виглядає наступним чином:

$$sum : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$mult : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$\forall a,b,c \in \mathbb{Z} \cdot mult(sum(a,b),c = sum(mult(a,c),mult(b,c))$$

Тобто, нічим не відрізняється від специфікації абстрактного типу даних. Треба визнати, що програмісти повторюють шлях математиків дев'ятнадцятого та початку двадцятого століть.

Далі, між алгебраїстами і Б. Меєром з одного боку і розробниками методу RAISE з іншого, можна виявити наступну відмінність: перша сторона, на відмінну від другої, специфікує відносини не тільки між парами функцій стану і функцій виходу, а також і між двома функціями стану. В роботі [15] Б. Меєр дає аксіому для пари функцій стану (non-derived generator) remove і put. У представлених вище аксіомах для арифметики Пеано також є аксіома для спільного опису додавання sum і ітератора suc. Зрозуміло, що обидві ці функції теж є функціями стану.

Крім того, при алгебраїчному підході ніхто не змушує задавати властивості функцій обов'язково попарно. Тобто функцій може бути одна (складання будь-якого числа з нулем дає нуль) або, скажімо, чотири. Так наприклад, визначаються функції синуса та косинуса:

$$sum(mult(sin(x),sin(x)),mult(cos(x),cos(x))) = 1$$

Тут символом mult позначається функція добутку дійсних чисел. Звичайно, такі складні рівності ускладнюють тестування. Налагодження та тестування потрібно починати з більш простих рівностей.

Відповідь на питання, чи є достатньо повною така аксіоматична специфікація типу, яка використовується в методах формальної розробки ПЗ, залишимо на авторитеті Б. Меєра та J. Guttag-а: «A detailed look at sufficient-completeness is contained in Guttag».

Специфікація арифметики Пеано мовою формальних специфікацій Z

Проєкт із програмуванням арифметики Пеано є невеликим. Мова Z [19] використовується у ньому з наміром освоєння мов формальних специфікацій у закінчених проєктах.

Це полегшує необхідність робити під час розробки, на думку Д. Парнаса, «невеликі кроки, щоб забезпечити відповідність між абстрактним уявленням користувачів про систему і конкретним працюючим кодом» [17]. ASCII-версія специфікації арифметики Пеано очевидно може бути записана мовою формальних специфікацій Z [3;19]. Початкову ASCII – версію специфікації, придатну для безпосереднього використання в коментарях коду програм, утилітою ZTC [13], можна відконвертувати в LaTex-версію, у форму звичну для математиків.

Приклад ASCII-версії специфікації з аксіомами визначення функцій suc і sum:

```
spec
   generic[T]
      one: T;
      suc: T fun T;
      sum: T & T fun
   where
      forall a,b: T @ sum(a,one) = suc(a)
          and sum(a,suc(b)) = suc(sum(a,b))
      end generic
      end spec
```

Відкомпільована в LaTex версія специфікації виглядає так:

one : T
suc : T → T
sum : T × T → T
∀ a, b : T · (sum(a, one) =
suc(a) ∧ sum(a, suc(b)) =
suc(sum(a, b))

Отримана специфікація повністю збігається з рекомендаціями Б. Меєра [15].

Обґрунтування існування алгоритму для суматора Грассмана

Перед тим як розпочати розробку поставленої задачі, важливо переконатися, що необхідний алгоритм існує і може бути реалізований (чого зазвичай програмісти не роблять). Запропонований в статті проєкт начебто створений для застосування частково-рекурсивних функцій Черча на практиці. Ітератор Пеано збігається з найпростішою функцією Черча з тою ж назвою suc – це перше правило з визначення суматора Грассмана. Друге правило записується через композицію ітератора та проєктування кортежу з трьох елементів на третю координату. Значить правило може бути виражене через елементарні операції над функціями і задовольняє вимогам рекурсії:

$$\mathrm{sum}(x,\mathrm{suc}(k)) = \mathrm{suc}(\mathrm{sum}(x,k)) = \mathrm{suc}\left(pr_3^3(x,k,\mathrm{sum}(x,k))\right)$$

Використовуючи примітивну функцію проєктування кортежу з 3 чисел на третій співмножник – pr_3^3, формулу обчислення суматора вдалося подати у вигляді оператору рекурсії Черча. Відповідно суматор можна запрограмувати.

Алгоритм ітератора

При описі алгоритму використовуються наступні позначення:

- dgts – це множина цифр від 0 до 9.
- seq1 – позначення типу непустих послідовностей.
- front – уся послідовність без останнього елемента.

– last – останній елемент послідовності.
– функція suc – на непустій послідовністю цифр (seq1 dgts) будує іншу.
– Послідовності записуються за допомогою подвоєних кутових дужок – << … >> в ASCII-стилі.
– Функція suc1 обчислює наступне значення останнього елемента послідовності і повертає пару: послідовність довжиною 1 із, можливо, зміненої цифри << n + a >> та ознаки переповнення.
– Функція suc2 – це крок рекурсії.

Функція suc змінює останній елемент послідовності і застосовує себе до меншої непустої послідовності без останнього елемента, з можливо зміненою ознакою переповнення. Символ ^ означає операцію конкатенацію послідовностей. Якщо послідовність була точно з одного елемента, то у разі переповнення до зміненого останнього елемента спереду додається 1.

```
---- File:./z/suc.zsl
spec
  input suc0.zcl
  schema suc
    suc : seq1  dgts          fun   seg1 dgts;
    suc2 : seq1  dgts  &0..1 fun   seg1 dgts;
    suc1 : gts           &0..1 fun  dgts &0..1;
  where
    forall s : seq1  dgts@  suc(s) = suc2(s,0);
    forall s : seq1  dgts;a : 0..1@
    suc2(s,a) = (let na == suc1(last s,a);n1 == ≪ first na ≫;a1 == second na  @ if  front s ≠ ≪≫
              then suc2(front s,a1)   ^n1
              else if  a1 = 0
              then n1
              else ≪ 1 ≫ ^ n1
              );
    forall n : dgts;          a : 0..1@
              suc1(n,a) = if a = 1 then
              if n + a > 9  then (0,1)  else (n + a,0)
              else (n,0)
  end schema
  end spec
---- End Of File:./z/suc.zsl
```

Специфікація алгоритму ітератора на LaTex:

```
dgts : P N
dgts = 0 . . 9

suc
suc : seq1 dgts → seq1 dgts
suc2 : seq1 dgts × 0 . . 1 → seq1 dgts
suc1 : dgts × 0 . . 1 → dgts × 0 . . 1
∀ s : seq1 dgts · suc(s) = suc2(s, 0)
∀ s : seq1 dgts; a : 0 . . 1 · suc2(s, a) =
   (let na == suc1(last s, a); n1 == first na ; a1 == second na · if
      front s /= then suc2(front s, a1) ^ n1
        else (if a1 = 0 then n1 else 1^ n1))
∀ n : dgts; a : 0 . . 1 · suc1(n, a) =
    if a = 1 then (if n + a > 9 then (0, 1) else (n + a, 0))
     else (n, 0)
```

Ця спроба описати алгоритм мовою Z виглядає незручною і марною, бо в результаті програмування текст виявився чи не довшим за текст методу [5]. Відсутність аналога змінних і виразів подібних до операторів циклів (які присутні в мові формальних специфікацій RAISE (далі – RSL) [22] ускладнює практичну роботи програмістів і робить її менш зручною порівняно з RSL.

Трохи зручніша версія специфікації ітератора може виглядати наступним чином:

```
dgts : P N
─────────────
dgts = 0 . . 9

inc == ( λ old, new : dgts · if (old = 9 ∧ new = 0) then 1 else 0)

mdf == (λ s : seq1 dgts; i : 0 . . 1 · if i = 1 then 0 ^a s else s)

suc ──────────────────────────────────
suc : seq1 dgts → seq1 dgts
──────────────────────────────────
∀ s : seq1 dgts ·
∀ i : 1 . . #(suc(s)) − 1 ·
   let ms == mdf(s, #(suc(s)) − #s) ·
   (last(suc(s)) = (last s + 1) div 10
   ∧ suc(s)(i) = ms(i) + inc(ms(i + 1), suc(s)(i + 1)))
──────────────────────────────────
```

Як і у випадку схеми для факторіалу цикл замінюється синтаксичною конструкцією з квантором загальності. Короткі функції визначаються через лямбда – вирази. Функція inc визначає наявність переповнення, воно виникає якщо вихідна цифра дорівнювала цифрі 9, а нова – дорівнювала цифрі 0. Функція mdf змінює вихідну послідовність цифр, дописуючи їй лідируючий нуль. Це потрібно зробити, якщо нова послідовність виявиться довшою за вихідну. Така ситуація виникає, коли вихідна послідовність починалася з цифри 9. ms – це позначення можливо подовженої вихідної послідовності, яке вводиться конструкцією let. Тобто, ms це застосування функції mdf до вихідної та різниці довжин вихідної та нової suc(s) послідовностей. В результаті послідовності ms і suc(s) гарантовано мають однакову довжину. Причому остання цифра у suc(s) рахується додаванням 1 до останньої цифри послідовності s та діленням на 10. А кожна наступна цифра нової послідовності suc(s)(i) визначається як сума поточної цифри старої та результату переповнення на попередньому (i + 1) кроці.

Аксіоматика комутативної напівгрупи мовою Z

Аксіоматика комутативної напівгрупи виглядає наступним чином [1]:

```
[T] ═══════════════════════════
  f : T × T → T
──────────────────
∀ x, y : T · (∃1 z : T · f(x, y) = z)
∀ x, y, z : T · f(x, f(y, z)) = f(f(x, y), z)
∀ x, y : T · f(x, y) = f(y, x)
──────────────────
```

Множина T з функцією f називається комутативною напівгрупою, якщо виконуються умови:

1. f – всюди визначена (стрілка, яка використовується в сигнатурі функції є символом всюди визначеної функції);
2. функція f задовольняє умову асоціативності;
3. функція f задовольняє умову комутативності.

Множина чисел Пеано та функція суматора Грасмана, згідно доведеного вище, задовольняють першу умову визначення напівгрупи. Докладно про доведення асоціативності та комутативності суматора Грассмана описано в роботі [5].

Таким чином, реалізований у проєкті [5] клас natural можна вважати напівгрупою натуральних чисел за додаванням.

Функція віднімання

Після обговорення суматора потрібно було розглянути формули для порівняння двох чисел (менше, рівно, більше), сформувати двосторонній натуральний ряд, та ввести операцію віднімання. Для цілей дослідження успадкування в поточному проєкті була реалізована функція віднімання, способом схожим на порівняння Грассмана з використанням ітератора. В силу того, що функція не є повністю визначеною на натуральних числах, вона не є операцією. Для деяких двох чисел a і b третє число, яке задовольняє умові sum(c, b) = a), буде називатися різницею і позначатися знаком dif(a, b). І це означає введення частково певної функції, яка буде називатися віднімання:

$$\mathrm{dif} : T \times T \mapsto \mathrm{sum}(\mathrm{dif}(a, b), b) = a$$

Різниця визначена не для всіх елементів з натуральних чисел, тому функція виявляється частково визначеною і позначається іншою функціональною стрілкою. Додаткові вимоги для специфікації напівгрупи натуральних чисел виглядають наступним чином:

$$[T]$$

$$\mathrm{sum} : T \times T \to T$$

$$\mathrm{dif} : T \times T \mapsto T$$

$$\forall x, y : T \cdot (\exists_1 z : T \cdot \mathrm{sum}(x, y) = z)$$

$$\forall x, y, z : T \cdot \mathrm{sum}(x, \mathrm{sum}(y, z)) = \mathrm{sum}(\mathrm{sum}(x, y), z)$$

$$\forall x, y : T \cdot \mathrm{sum}(x, y) = \mathrm{sum}(y, x)$$

$$\forall x, y : T \cdot (x, y) \in \mathrm{dom\ dif} \Rightarrow \mathrm{sum}(\mathrm{dif}(x, y), y) = x$$

Далі, саме функція віднімання в обох можливих спробах наслідування (як у бік звуження домену класу – від групи до напівгрупи, так і у бік розширення домену класу – від напівгрупи до групи [6]) призводила до порушення в роботі поліморфних функцій. Спільним між цим прикладом і прикладом Р. Мартіна є те, що віднімання є функцією стану, а домен її класу знаходиться з лівої і правої сторін функціональної стрілки.

Тестування

Подробиці з розробки та тестування класу natural викладені в роботі [5]. Зокрема, алгебраїчні рівності sum(a, one) = suc(a) та sum(a, suc(b)) = suc(sum(a, b)) були використані для розробки відповідних драйверів тестів [5]. У цих тестах просто підставляються різні значення для пошуку першої відмови. Програмування цих тестів виглядає значно простіше, ніж розробка тестів за технологіями чорного та/або білого ящиків [16]. При цьому важливо зазначити, що створення алгебраїчних тестів не є мистецтвом, а є досить механічним прийомом.

Можна відзначити, що тести з використанням алгебраїчних рівностей широко використовуються в монографії W. Cody і W. Waite-а [9] для роботи з функціями дійсного аргументу.

Висновки

– Алгебраїчний підхід до розробки та тестування ПЗ виглядає досить універсальним та перспективним засобом.

– Властивості функцій типу даних (як і методів класу), які задаються алгебраїчними рівностями не враховуються компіляторам при застосуванні універсального поліморфізму включення.

– Наявність функції стану (і відповідного методу в класі) у типі даних робить небажаною зміну домену класу. Поява домену класу одночасно з лівої та з правої сторони функціональної стрілки в силу одночасної ко- та контра-варіантності функціонального типу дозволяє успадкування, але робить небезпечним використання універсального поліморфізму включення

– Алгебраїчний підхід дозволяє перейти від мистецтва (в сенсі ‘Мистецтво програмування для ЕОМ’ Д. Кнута і ‘Мистецтво тестування програм’ Г. Майєрса) до досить простих, практично технічних прийомів програмування.

– Наявність у мові RSL синтаксичних конструкцій (Repetitive Expressions) для багаторазового повторення операторів дозволяє робити на мові RSL ‘невеликі кроки, щоб забезпечити відповідність між абстрактним уявленням про систему і конкретним кодом’. Таким чином, специфікації на мові RSL є більш зрозумілими та близькими до практичного кодування, ніж на мові Z.

Список використаних джерел:

1. Піскунов О.Г. Типи, множини та класи. 2011. С. 19. URL: https://www.researchgate.net/publication/334174126 (дата звернення: 01.02.2024).

2. Піскунов О.Г. Про відмінності між поняттями типу та. *Вісник Київського національного університету імені Тараса Шевченка*. Серія : Фізико-математичні науки. 2015. № 3. С. 106–114.

3. Піскунов О.Г. LaTex та вимоги державного стандарту. 2022. С. 74. URL: https://www.researchgate.net/publication/359860334 (дата звернення: 01.02.2024).

4. Піскунов О.Г., Жултинська А.К. Документування процесу розробки програмного забезпечення. 2024. С. 324. URL: https://www.researchgate.net/publication/377261513 (дата звернення: 01.02.2024).

5. Піскунов О.Г., Рудик В.І., Петренко І.А. Арифметика Пеано: від специфікації до класу. 2022. С. 45. URL: https://www.researchgate.net/publication/365979331 (дата звернення: 01.02.2024).

6. Піскунов О.Г., Мічуда А.М. Переозначення додавання: небезпечне наслідування в групі цілих. 2023. С. 36. URL: https://www.researchgate.net/publication/366867037 (дата звернення: 01.02.2024).

7. Cardelli L., Abadi M. A theory of objects. New York: Springer-Verlag, 1996. P. 396.

8. Cardelli L. A semantics of multiple inheritance. Information and Computation. 1988. № 76. P. 138–164.

9. Cody W., Waite W. Software manual for the elementary functions. New Jersey: Prentice-Hall, 1980. P. 289.

10. Date, C.J. An Introduction to Database Systems, 7th Edition. London, UK: Addison-Wesley, 2000. 938 p.

11. Guttag J.V. Abstract Data Types and the Development of Data Structures. Communications of the ACM. 1977. Vol. 20. № 6. P. 396-404.

12. Haxthausen A. Lecture Notes on The RAISE Development Method. Kongens Lyngby: DTU, 1999. P. 20.

13. Jia X. ZTC: A Type Checker for Z Notation. User's Guide (Version 2.03).Chicago: DePaul University, USA, 1998. P. 44.

14. Martin R. Clean Architecture: A craftsman's guide to software structure and design. Boston, U.S: Prentice-Hall, 2018, 378 p.

15. Meyer B. Object-Oriented Software Construction. Second Edition. London: Pearson Education, 2022. P. 1024.

16. Myers G., Sandler C., Badgett T. The art of software testing, 3rd ed. New Jersey, USA:J. Wiley & Sons, Inc, 2012. 240 p.

17. Parnas D.L. Really rethinking 'formal methods'. New York, U.S: IEEE Computer Society, Computer, 2010, N 43, pp. 28–34.

18. Piskunov A.G. Inheritance of Abstract Automata. *Вісник Київського національного університету імені Тараса Шевченка*. Серія : Кібернетика. 2011. № 11. С. 40-44.

19. Spivey J.M. The Z Notation: A Reference Manual, 2nd edition. New Jersey: Prentice Hall International Series in Computer Science, 1992. P. 158.

20. Stepanov A.A., Ros D.E. From Mathematics to Generic Programming. London, UK: Addison-Wesley, 2015, 285 p.

21. Wing J., Liskov B. Family Values: A Behavioral Notion of Subtyping. Pittsburgh,U.S: ACM, ACM Trans. Program. Lang. Syst., 16(6), 1994. pp 1812-1841 (дата звернення: 01.02.2024).

22. The RAISE Language Group. The RAISE SPECIFICATION LANGUAGE. Kongens Lyngby, Denmark: Prentice Hall Europe, 1992, 396 p.

23. Guttag, J.V. and Horning, J.J., The algebraic specifications of abstract data types. URL: https://www.semanticscholar.org/paper/The-algebraic-specification-of-abstract-data-types-Guttag-Horning/e4c8b1db0c839a07a833db51c5ac00e6ffd5a922 (дата звернення: 01.02.2024).

References:

1. Piskunov O.G. (2011). Typy, mnozhyny ta klasy [Types, sets and classes]. *ResearchGate, P. 19.* Retrieved from https://www.researchgate.net/publication/334174126.

2. Piskunov O.G. (2015). Pro vidminnosti mizh poniattiamy typu ta klasu [On the differences between the concepts of type and class]. *Visnyk Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. Seriia : Fizyko-matematychni nauky – Bulletin of Taras Shevchenko Kyiv National University. Series: Physical and mathematical sciences,* 3, *106–114* [in Ukrainian].

3. Piskunov O.G. (2022). LaTex ta vymohy derzhavnoho standartu. LaTex and the requirements of the state standard. *ResearchGate, P.* 74. Retrieved from https: //www.researchgate.net/publication/359860334.

4. Piskunov O.G., Zhultynska A.K. (2024). Dokumentuvannia protsesu rozrobky prohramnoho zabezpechennia [Documentation of the software development process]. *ResearchGate, P.* 324. Retrieved from https://www.researchgate.net/publication/377261513.

5. Piskunov O.G., Rudyk V.I., Petrenko I.A. (2022). Aryfmetyka Peano: vid spetsyfikatsii do klasu [Peano Arithmetic: From Specification to Class]. *ResearchGate, P.* 45. Retrieved from https://www.researchgate.net/publication/365979331.

6. Piskunov O.G., Michuda A.M. (2023). Pereoznachennia dodavannia: nebezpechne nasliduvannia v hrupi tsilykh [Redefining Addition: Dangerous Imitation in a Group of Integers]. *ResearchGate, P.* 36. Retrieved from: https://www.researchgate.net/publication/366867037.

7. Cardelli L., Abadi M. A. (1996). Theory of objects. New York, U.S.: Springer-Verlag.

8. Cardelli L. A. (1988). Semantics of multiple inheritance. *Information and Computation*,76.

9. Cody W., Waite W. (1980)/ Software manual for the elementary functions. New Jersey, U.S.: Prentice-Hall.

10. Date, C.J. (2000). An Introduction to Database Systems, 7th Edition. London, UK: Addison-Wesley.

11. Guttag J.V. (1977). Abstract Data Types and the Development of Data Structures. *Communications of the ACM,Vols. 20, 6, 396-404.*

12. Haxthausen A. (1999). Lecture Notes on The RAISE Development Method. Kongens Lyngby: DTU.

13. Jia X. (1998). ZTC: A Type Checker for Z Notation. User's Guide (Version 2.03).Chicago, U.S.: DePaul University.

14. Martin R. (2018). Clean Architecture: A craftsman's guide to software structure and design. Boston, U.S: Prentice-Hall.

15. Meyer B. (2022). Object-Oriented Software Construction. Second Edition. London, UK: Pearson Education.

16. Myers G., Sandler C., Badgett T. (2012). The art of software testing, 3rd ed. New Jersey, USA:J. Wiley & Sons, Inc.

17. Parnas D.L. (2010). Really rethinking 'formal methods'. *IEEE Computer Society, Computer, 43, 28–34.*

18. Piskunov A.G. (2011). Inheritance of Abstract Automata. *Visnyk Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. Seriia : Kibernetyka – Bulletin of Taras Shevchenko Kyiv National University. Series: Cybernetics, 11, 40-44.*

19. Spivey J.M. (1992). The Z Notation: A Reference Manual, 2nd edition. New Jersey, USA: Prentice Hall International Series in Computer Science.

20. Stepanov A.A., Ros D.E. (2015). From Mathematics to Generic Programming. London, UK: Addison-Wesley.

21. Wing J., Liskov B. (1994). Family Values: A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst., 16(6), 1812-1841.*

22. The RAISE Language Group (1992). The RAISE SPECIFICATION LANGUAGE. Kongens Lyngby, Denmark: Prentice Hall Europe.

23. Guttag, J.V. and Horning, J.J. The algebraic specifications of abstract data types. Retrieved from https://www.semanticscholar.org/paper/The-algebraic-specification-of-abstract-data-types-Guttag- Horning/e4c8b1db0c839a07a833db51c5ac00e6ffd5a922.