DOI 10.37882/2223-2966.2024.03.32

ВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ФУНКЦИОНАЛЬНЫХ ЯЗЫКОВ ДЛЯ АВТОМАТИЗАЦИИ ЗАДАЧ ВЕРИФИКАЦИИ

POSSIBILITY OF USING FUNCTIONAL LANGUAGES TO AUTOMATE VERIFICATION TASKS

I. Sokolov

Summary. Interpreted programming languages with dynamic typing allow you to write programs and even entire systems quickly enough. However, the problems of specification and verification of programs written in such languages are practically not amenable to automation and are often performed manually. In this paper, a method of automated verification of programs written in languages with dynamic typing is proposed. To solve problems with types, the program is translated into a functional programming language that is used as a specification language with subsequent parameterization of arguments and the construction of a full-fledged model for property-based testing.

Keywords: program verification, testing, functional programming languages, specification, formal methods, property-based testing.

Соколов Илья Николаевич
Аспирант, Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики
sokolov@orfun.ru

Аннотация. Интерпретируемые языки программирования с динамической типизацией позволяют достаточно быстро писать программы и даже комплексы программ. Однако проблемные вопросы спецификации и верификации программ, написанных на таких языках практически, не поддаются автоматизации и зачастую выполняются вручную. В данной статье предложен метод автоматизированной верификации программ, написанных на языках с динамической типизаций. Для решения вопросов с типами программа транслируется в функциональный язык программирования который используется в качестве языка спецификации с последующей параметризацией аргументов и построением полноценной модели для property-based тестирования.

Ключевые слова: верификация программ, тестирование, функциональные языки программирования, спецификация, формальные методы, property-based testing.

Введение

Процесс спецификации является важным элементом для многих процессов, связанных с разработкой программного обеспечения. Спецификация представляет собой детальный набор требований к программному обеспечению (ПО) [1] и используется как для документирования, так и для автоматизации процессов тестирования или оптимизации. Автоматизировать процесс генерации тестовых наборов для верификации программного обеспечения возможно с использованием имеющейся спецификации [2]. Спецификация при этом может быть написана вручную, или сгенерирована автоматически. Сложность автоматической генерации спецификации, будет при этом напрямую завесить от метода типизации в используемом языке программирования. В случае использования языка со статической типизацией, спецификация может быть получена в том числе из сигнатур функций, классов, методов или иных элементов языка, а также эффективно дополнена средствами статического или динамического анализа кода [3]. Большинство ошибок в таких языках обнаруживаются в процессе компиляции программы.

При написании программ на языках программирования (ЯП) с динамической типизацией, получение спецификации программ может быть затруднительно [4]. В таких языках нет необходимости прямого указания типов при объявлении переменных или аргументов функции. Проверка типов при этом производится во время фактического исполнения операций и получить ошибку в таких языках можно просто если перед операцией не привести переменную к нужному типу вручную.

Статический анализ кода может выявить лишь некоторые ошибки, так как процесс анализа подразумевает только проверку по различным шаблонам. Эффективными могут оказаться инструментальные средства «динамического анализа». В таком случае фрагменты кода запускаются в какой-либо среде [4]. В том числе, существуют подходы с использованием методов машинного обучения, которые позволяют имитировать исполнение программы и получать отчет о процессе ее исполнения [5], однако отмеченные выше вопросы при этом лишь частично разрешаются. Сложности спецификации в языках с динамическими типами зачастую влекут за собой вопросы, связанные с верификацией. При этом ошибку допустить гораздо проще чем при программировании на языках с типизацией статической. Этот факт увеличивает нагрузку в вопросах написания модульных тестов.

Функциональные языки программирования менее популярны в разработке, однако отличаются сильной и статической типизацией. В программах на таких языках, как и в языках с динамической типизацией нет необходимости явно указывать типы. Для определения типов в таких языках анализируется тело функции (абстракция) и операции над переменными [6].

Существует объективное предположение, что языки программирования могут быть использованы в качестве языка спецификации [7] и можно предположить, что именно функциональные языки наиболее эффективны для в этих целях, особенно для программ, написанных на языках с динамической типизацией. Функциональная парадигма предоставляет возможность представления программы как математического выражения [8]. Для каждой функции можно задать параметры проанализировав внутренние операции и получить достаточно детальную спецификацию, к которой могут быть применены автоматизированные методы генерации тестовых наборов и верификации.

Целью исследования, результаты которого представлены далее, является оценка применимости функциональных языков программирования как языков спецификации для последующей верификации ПО.

Для достижения поставленных целей:

1. будет предложен синтаксис функционального ЯП на базе типизированного лямбда исчисления со строго одноместными функциями;
2. рассмотрены варианты использования такого языка для спецификации программ, написанных с использованием динамических языков программирования;
3. предложен метод автоматизированной верификации транслированной программы

Краткий обзор предметной области

В качестве предметной области для целевого исследования выступают процессы спецификации и верификации программного обеспечения, написанного на языках программирования с динамической типизацией. Далее рассмотрим базовые направления и понятия, характеризующие эту область.

Верификация в общем случае подразумевает под собой исследование системы по стратегиям «Белого», «Серого» и «Черного» ящиков. Стратегия «Черный Ящик» подразумевает исследование поведения системы через интерфейсы, которые она предоставляет. Стратегия тестирования «Белого Ящика» — подразумевает полный доступ к реализации системы и включает в себя методы анализа исходных кодов. Стратегия «Серого ящика», предполагает комбинирование этих двух вариантов, и предполагает использовать техники «Белого Ящика» для подготовки тестовых наборов для тестирования по стратегии «Черного Ящика». Такой подход является наиболее выгодным с точки зрения автоматизации генерации тестовых наборов. В контексте процессов тестирования спецификация необходима для построения модели программного обеспечения.

Модель ПО — это формальное описание программы как набора функций, их взаимодействий и свойств. Для построения таких моделей программные коды могут быть проанализированы статически или динамически [2].

Статический анализ — анализ кода программы без её выполнения. Такой подход позволяет найти ошибки и уменьшить количество дефектов в программном приложении [4].

Динамический анализ подразумевает исполнение кода в виртуальной машине, перекомпиляцию кода. [3]

Среди методов тестирования выделяется метод **property-based**, который подразумевает автоматическую верификацию функции, на основании предварительно описанной спецификации параметров. Данный метод тестирования используется в инструменте QuickCheck.

Пример параметризации, приведенный в документации пакета pytest-quickcheck:

```
@pytest.mark.parametrize(«prime», [2, 3, 5])
@pytest.mark.randomize(i1=int, f1=float, ncalls=1)
def test_gen_parametrize_with_randomize_int_float (prime, i1, f1):
 pass
```

Методы исследования

Для исследования применимости функциональных языков для спецификации ПО в контексте настоящей статьи, будет приведен синтаксис ЯП на базе типизированного лямбда исчисления, основным объектом в котором будут являться одноместные функции высшего порядка. Таким образом каждая транслированная функция будет представлена в виде:

λ a. λ b. λ c. a что эквивалентно функции Python: def f(a, b, c): return a

В Функциональных программах, где программы рассматриваются как математическое выражение, существует единственная точка начала вычисления. Таким образом, к каждой атомарной функции может быть применен метод параметризации с использованием метода «тройки Хоара», коллекций предикатов предварительных условий и пост условий могут быть применены для генерации тестовых наборов.

Логика Хоара

Логика Хоара представляет собой формальную систему доказательства корректности компьютерных программ.

Основной характеристикой логики Хоара является «Тройка Хоара»:

{P} C {Q}

Где **P** и **Q** являются утверждениями, а **C** — командой.

P — описывает состояние контекста до применения команды, а **Q** описывает состояние после выполнения команды **C** [9].

1. Для пустых команд, предусловия и постусловия будут совпадать
2. Цепочка предусловий и постусловий в случае с использованием функционального языка будет формироваться от точки редекса до сигнатуры объявления функции
3. Операторы ветвления являются для последующих блоков предусловиями, а их отрицание, является предусловием для блока **else**. Постусловия полностью формируются из соответствующего блока.

Частичная корректность подразумевает что мы можем доказать, что операция начнет выполнятся. **Полная корректность** будет требовать доказательства полного исполнения операции [9].

Функциональные языки программирования

В качестве языка формальной спецификации, будет рассмотрен функциональный язык на базе типизированного лямбда исчисления. Синтаксис будет основан на абстракциях и аппликациях одноместных функций. Функции могут быть переданы как аргумент и возвращены как значения. Процесс выполнения трактуется как выполнение значений функций в математическом понимании. Теоретической базой для функциональных языков является лямбда-исчисление, которое в свою очередь основано на комбинаторной логике.

В комбинаторной логике основой вычислений является одноместная функция высшего порядка и операция аппликации. Так как функция в данном случае может использоваться в качестве аргумента для функции, любую многоместную функцию можно свести к одноместным [10].

Минимально необходимый набор комбинаторов:
1. I x = x
2. K x y = x
3. S x y z = x z (y z)

Более сложные комбинаторы могут быть выражены через данные комбинаторы, например комбинатор неподвижной точки:

Y = S (K (S I I)) (S (S (K S) K) (K (S I I)))

Который может быть использован в вызовах по значению. В частности, для рекурсивных вызовов функций.

На основе комбинаторной логики, помимо Лямбда исчисления основаны парадигмы программирования, в которых при объявлении функций не используются промежуточные переменные, но составляются цепочки из вызовов функций. Таким образом язык комбинаторной логики позволяет полностью избавиться от переменных, используя при этом комбинаторы.

Лямбда исчисление — это формальная система, в основе которой лежат две операции: Аппликация и Абстракция. В чистом лямбда исчислении функции рассматриваются как правила, а не как графики [11], изучается аппликативное поведение функций.

Абстракция позволяет описывать функции независимым образом.

Функциональные языки программирования, описываются на базе лямбда-исчисления. Так как воплощает такой способ определения и применения функций в наиболее чистой форме. В лямбда исчислении все является функциями: аргументы, которые функции принимают тоже функции, и результат, который возвращают функции — тоже функции [12]

Пример: (Λx.x^2 + 1) (3) =10

Программа, написанная на функциональном языке, представляет собой некоторое выражение, а выполнение программы означает вычисление значения данного выражения. Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом. Многие проблемы, возникающие в программировании, предстают в Λ исчислении в чистом виде. [11]

Пример: Алгоритм «быстрой сортировки» Haskell:

```
let qs (x: xs) = qs [y | y <— xs, y < x] ++ [x] ++ qs [y | y <— xs, y >= x]
```

Для определения типов в данных в функциональных языках с параметрическим полиморфизмом используется **алгоритм Хиндли-Милнера**, который позволяет определять типы без их явного определения. Типы в таком случае определяются посредством исследования тела функции и операций над аргументами.

Выводы

Проблема с определением динамических типов может быть решена несколькими способами. Например, это может быть анализ сегментов памяти, в которых хранится значение переменной. В случае трансляции в функциональный язык все переменные исходной программы на ЯП с динамической типизацией могут быть приведены к строгим типам. Более того если параметризировать такую программу, то на базе спецификации может быть построена модель для последующей верификации.

Параметризация в данном случае может быть произведена в процессе динамического анализа над программой в транслированном виде, а для генерации тестовых наборов может быть использована комбинация методов РВТ. Динамический анализ такой программы сам по себе может быть более простым, если использовать функциональный язык с одноместными функциями высшего порядка. В таком случае каждая операция будет атомарной, в том числе это позволит работать с любым поддеревом вызова, в том числе способом символьного исполнения.

Результаты

Для демонстрации полезных семантических свойств функциональных ЯП в задачах верификации, будет приведен синтаксис такого языка на базе типизированного лямбда исчисления с использованием одноместных функций высшего порядка. Синтаксис языка содержит минимально необходимое количество термов, и обладает стандартным набором логических и арифметических операций. Количество типов в языке также сведено к минимуму. Правила вычислений для термов и правила вывода типов можно найти источниках [10]. Для используемого языка использовалось описание из книги «Типы в языках программирования» [12]. Язык, описанный в данной работе, отличается тем, что функции строго одноместны и есть отличия в синтаксисе некоторых термов.

Язык оперирует функциями высшего порядка, которые могут быть переданы в качестве аргумента и возвращены в качестве результата. В языке отсутствует оператор присваивания и переменные. Для упрощения последующего анализа, функции для многих аргументов я языке также не предусмотрены, так как функции нескольких переменных всегда можно свести к функциям одноместным [10]. Программы на этом языке представлены в максимальной декомпозиции аргументов. Легко выявляется связь одних аргументов с другими. Типизация в языке строгая и сильная, для определения типов используется алгоритм Хиндли-Милнера [6].

Синтаксис языка

Логические операторы:
"=" эквивалентность, «>» больше, «<» меньше, «|» — или, «&» — И,
"!" — отрицание

Константы: true — истина, false— ложь, 0 — ноль.

x	Аргумент функции
lam t -> t	абстракция
lam t:T -> t	Типизированная абстракция
~(t)	Комбинатор неподвижной точки
t(t)	аппликация
{t, t}	пара
t.1	первая проекция
t.2	вторая проекция
cons [t, t]	Конструктор списка
nil[T]	Пустой список
head[T]	Первый элемент списка
tail[T]	Последующие элементы списка
error	Исключение
t ? t1 : t2	условное выражение
t ? t1 | t2 ? i2	Case (тавтология)
++ t	следующее число
-— t	предыдущее число
iszero t	Проверка на ноль
isexist L t	Проверка существующего элемента по индексу
t Is T	Проверка на принадлежность переменной к типу

Множество базовых типов: { Int, Float, Str, Bool, Pair, List}

Пример трансляция с динамическими типами

```
def f(n):
        if isinstance (n, list):
 return n[0] + 1
        return n+1
```

В транслированном виде:

```
lam ni ->
        lam nl -> iszero nl
        ? n +1
        : head nl +1
```

где ni — n типа int, a nl — имеет тип списка.

Пример трансляции «Факториал»

Рекурсивные вызовы функций реализованы через комбинатор неподвижной точки (терм ~): ~(x) = lam x ->x (f(x)).

```
def f(n):
    if n<1:
    return -1
    return n * f(n-1)
```

Трансляция:

```
lam n -> n<1? 1: n * ~(n-1)
```

Параметризация «Тройки Хоара»

Так как каждая функция в данном языке предусматривает наличие лишь одного аргумента, а все вложенные блоки оформляются как отдельно взятая функция, параметризация будет осуществляться для каждой функции в контексте ее единственного аргумента.

{Px}Lam x -> {Py}lam y -> {Pz} lam z -> {Pz} 1 {Qz}{Qy}{Qx}

В свою очередь сумма множеств параметров будет описывать исходную функцию на ЯП с динамической типизацией:

{Px ∪ Py ∪ Pz} **fn(x, y, z)** {Qx ∪ Qy ∪ Qz}

В качестве примера приведена функция, которая возвращает квадраты всех четных элементов списка:

```
def f(l):
r = []
for x in l:
if x % 2 == 0:
r += x ** 2
return r
```

Результат трансляции:

```
lam ll -> head ll /2 is Int
 ? cons head ll * head ll ~(tail ll)
 : is zero ll ?:~(tail ll)
```

Параметризация:

Функция имеет две части вычислений, каждая из которых является функцией. Перед обоими функциями стоит условие **head ll /2 is Int**, соответственно это становится пред условием для части **cons head ll * head ll ~(tail ll)**, а отрицание этого условия , в свою очередь будет предусловием для **is zero ll ?:~(tail ll)**. Полное дерево построения предикатов приведено на Рис. 1

Таким образом на основании параметров пред и пост условий можно построить модель ПО. Наличие параметров для каждого аргумента позволит сгенерировать тестовые наборы для Данный функции. Более того предикаты в таком случае будут отражать связанность аргументов, что позволит сделать более точное описание модели.

Параметризация вложенных функций позволяет объединить требования для более общих функций, в контексте которых должны исполняться вложенные.

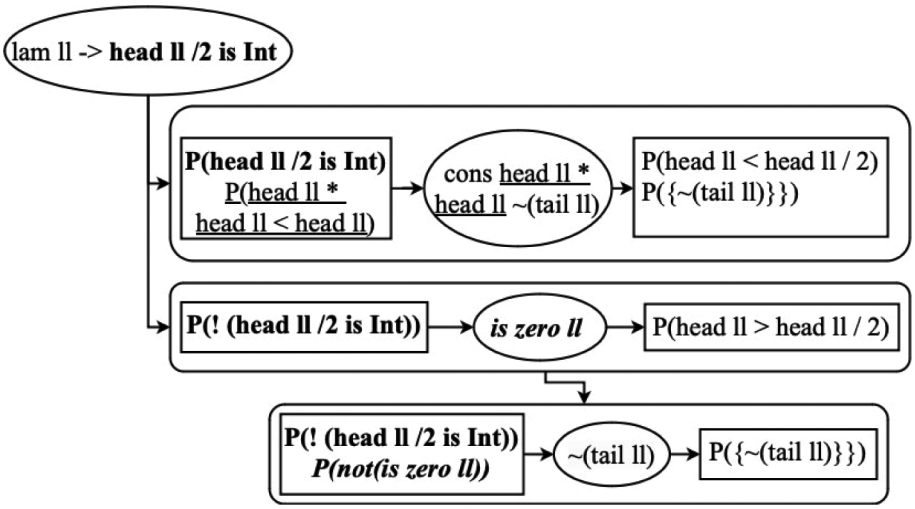

Рис. 1. Параметризация вложенных функций

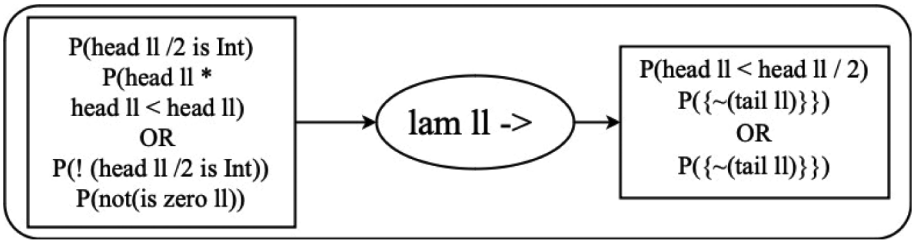

Рис. 2. Параметры для функции целиком

По контекстам параметров можно заключить

4. Входные данные — список целых чисел, который должен содержать четные элементы для того, чтобы получить непустой вывод
5. Над элементами списка выполняется операция умножения на себя

Построение тестовых наборов:

```
f([]) -> []
f[1,3,5] = []
f([1,2,3,4]) -> [4,16]
f([2.5, 3.6]) = []
f(["bad_type"]) = # TypeErr
f(["str_type", 2]) = # TypeErr
```

Данные тестовые наборы могут быть применены для верификации исходной функции.

В соответствии с полученной информацией о типах исходная функция может быть модифицирована с точным приведением типов.

Пример Модификация изначальной функции

```
def sum_plus_one(x, y, z):
    return x + y + z +1
```

Трансляция:

```
lam x ->
    x + lam y ->
            y + lam z ->
                    z + 1
```

Тестовые наборы:

```
sum_plus_one("bad_type", y, z) -> False
sum_plus_one (1,2,3) -> 7
```

Преобразованная форма исходной программы:

```
def sum_plus_one(x:int, y:int, z:int):
    return int(x) + int(y) + int(z) + 1
```

Выводы

В настоящей статье исследованы возможности использования функциональных языков программирования в качестве средства спецификации для последующего решения задач верификации ПО.

Рассмотрена предметная область и выделены известные методы автоматизации механизмов генерации тестовых наборов на базе спецификации ПО.

Предложен синтаксис функционального языка для его использования в качестве языка спецификации путём использования одноместных функций и минимальным набором термов и типов. Факт успешной трансляции в такой язык может свидетельствовать о том, что верифицируемая программа написана корректно и может быть выполнена. Предложен метод верификации программ, написанных на языках с динамической типизацией, основанный на синтаксисе и механизмах функционального программирования с одноместными функциями и тройками Хоара.

Приведены примеры трансляции, параметризации и верификации функций на языке Python. Предложенный метод может быть применен для верификации программ, написанных на языках программирования с динамической типизацией, и имеет ряд следующих далее полезных свойств.

— Исходная программа, написанная после трансляции представлена в виде строго типизированном виде. Это обстоятельство разрешает большое число вопросов, связанных с типами в таких языках как Python.
— Предоставляется возможность рассматривать исходную программу как последовательный вызов логически законченных функций одного аргумента. Метод позволяет произвести параметризацию и оценить связанность переменных в исходной программе, в том числе, автоматически применять такие механизмы как «Тройка Хоара»
— Отношение всех параметров вложенных функций, в достаточной мере описывает свойства исследуемой функции, которая может быть использована для генерации тестовых наборов методами «На базе модели» и, в частности «Property Based».
— Более того, программа на таком языке позволяет верифицировать любое поддерево дерева вызовов исходной.

Использование функциональных языков, открывает широкий спектр возможностей для анализа процесса исполнения программ, в том числе, что важно, на направлении тестирования программного обеспечения.

В частности, в функциональном программировании совокупность последовательных состояний вычислительного процесса представляется явным образом.

Следует отметить, однако, что в данной работе представлен ограниченный синтаксис языка, который, в том числе, не сможет в полной мере работать с классами и их объектами.

ЛИТЕРАТУРА

1. Habrias Henri, Frappier Marc. Software specification methods. — John Wiley & Sons, 2013.
2. Apfelbaum Larry, Doyle John. Model based testing // Software quality week conference / Citeseer. — 1997. — P. 296–300.
3. Bush William R, Pincus Jonathan D, Sielaff David J. A static analyzer for finding dynamic programming errors // Software: Practice and Experience. — 2000. — Vol. 30, no. 7. — P. 775–802.
4. Fairley Richard E. Tutorial: Static analysis and dynamic testing of computer software // Computer. — 1978. — Vol. 11, no. 4. — P. 14–23.
5. Fault-aware neural code rankers / Inala Jeevana Priya, Wang Chenglong, Yang Mei, Codas Andres, Encarnacion Mark, Lahiri Shuvendu, Musuvathi Madanlal, and Gao Jianfeng // Advances in Neural Information Processing Systems. — 2022. — Vol. 35. — P. 13419–13432.
6. Jones Mark P. From Hindley-Milner types to first-class structures // Proceedings of the Haskell Workshop / Citeseer. — 1995.
7. Parnas David Lorge. Really rethinking 'formal methods' // Computer. — 2010. — Vol. 43, no. 1. — P. 28–34.
8. Hudak Paul, Fasel Joseph H. A gentle introduction to Haskell // ACM Sigplan Notices. — 1992. — Vol. 27, no. 5. — P. 1–52.
9. Hoare Charles Antony Richard. An axiomatic basis for computer programming // Communications of the ACM. — 1969. — Vol. 12, no. 10. — P. 576–580.
10. Шалак ВИ. МИ Шейнфинкель и комбинаторная логика // Логические исследо— вания. — 2009. — no. 15. — P. 247–265.
11. Barendregt Hendrik P et al. The lambda calculus. — North-Holland Amsterdam, 1984. — Vol. 3.
12. Pierce Benjamin C. Types and programming languages. — MIT press, 2002.

Журнал «Современная наука: актуальные проблемы теории и практики»