УДК 004.42 doi: 10.15827/0236-235X.149.065-076 2025. Т. 38. № 1. С. 65–76

Система верифицируемых спецификаций программных компонентов с поддержкой встраивания и извлечения

П.А. Шапкин [1]✉

[1] Национальный исследовательский ядерный университет «МИФИ», г. Москва, 115409, Россия

Ссылка для цитирования
Шапкин П.А. Система верифицируемых спецификаций программных компонентов с поддержкой встраивания и извлечения // Программные продукты и системы. 2025. Т. 38. № 1. С. 65–76. doi: 10.15827/0236-235X.149.065-076
Информация о статье
Группа специальностей ВАК: 2.3.5
Поступила в редакцию: 06.03.2024 После доработки: 12.07.2024 Принята к публикации: 24.07.2024

Аннотация. Объектами данного исследования являются спецификация и верификация программных систем и их компонентов. Предмет исследования – унифицированный язык спецификаций, оснащенный соотнесением как с системами случайного тестирования, так и со средствами статической верификации на основе систем типов. Разнообразие языков программирования, систем конфигурирования, развертывания и другие инструменты требуют от разработчиков усилий по их интеграции. Упростить задачу помогает наличие верифицируемых спецификаций компонентов. В работе предложен подход к унифицированному представлению спецификаций, интегрированному с системами как для статической проверки типов, так и для динамического тестирования. Это решение опирается на методы аппликативных вычислительных систем и теории типов и предоставляет понятийный каркас для построения спецификаций, встраиваемых в различные программные среды. Недостаток возможностей статической верификации из-за ограниченности систем типов до некоторой степени устраняется за счет динамического тестирования. Тестирование осуществляется посредством интерпретации спецификаций в определения для систем случайного тестирования на основе свойств. Практическая значимость предлагаемого подхода состоит, в частности, в автоматизации процесса построения типизированных оберток, или фасадов, необходимых для использования компонентов из менее типизированных сред в языках программирования с более выразительными системами типов. Автоматизируются как верификация таких оберток, так и способы их построения за счет определения операций уточнения спецификаций. На практике это позволяет выявлять ошибки в типизации сторонних компонентов на ранних стадиях разработки. В статье приведены примеры спецификаций программ с побочными эффектами. В качестве основы для спецификаций использованы формализации из теории категорий. Проанализированы подходы к транслированию спецификаций в другие представления и к итеративному усовершенствованию спецификаций путем их трансформации.
Ключевые слова: программные системы, спецификация, верификация, семантика, логика, типизация

Введение. Важность задачи спецификации и верификации программных компонентов неуклонно растет, что обусловлено рядом тенденций реализации информационных технологий. Развитие *искусственного интеллекта* (ИИ) привело к тому, что уже более десятой доли программного кода генерируется автоматически, а неточная природа получаемых от ИИ решений означает необходимость создания и совершенствования средств спецификации задач и верификации сгенерированных компонентов. Появление и развитие новых платформ и подходов к вычислениям, таких как облачные технологии, блокчейн, дают толчок к созданию новых языков и инфраструктур программных компонентов, требующих интеграции между собой и с традиционными языками общего профиля [1], а значит, спецификации должны быть переносимыми и погружаемыми в различные вычислительные среды. Достижения в области логики и теории типов [2] обусловили создание мощных средств для построения статически верифицируемых программ, однако они сложны для использования программистами среднего или начального уровня, а прогнозируемое двукратное ускорение роста численности начинающих программистов вряд ли улучшит данную ситуацию. Существуют системы и языки, такие как TLA+ Why3, OpenJML, Frama-C, CPAchecker, специально предназначенные для описания и верификации спецификаций, а также подходы к верификации путем преобразования программного кода к представлению на языке спецификаций [3, 4], однако их применение не всегда возможно в силу высокой сложности полной трансляции кода программы. Более просты в использовании средства, верифицирующие спецификации путем динамического анализа, такие как Ortac [5], но обычно они являются узкоспециализированными.

В настоящей работе описывается подход к спецификации программных компонентов, реализованный автором в библиотеке Quasi-

Type и дающий в зависимости от возможностей целевой среды отчуждаемые спецификации, оснащенные средствами погружения в различные вычислительные среды для дальнейшей динамической или статической верификации. Целью является выполнение таких задач, как верификация решений, построенных на нетипизированных компонентах, взаимодействующих через интеграционное приложение на типизированном языке. В качестве примера может рассматриваться технология Scala.js, позволяющая использовать типизированный язык Scala для написания кода JavaScript (JS), нетипизированные внешние компоненты которого подключаются посредством типизированных оберток, или фасадов. Поскольку JS-код внешних компонентов недоступен для типизации, указанные в фасадах типы могут не соответствовать действительному поведению программы и требовать дополнительной верификации. При такой верификации объявленные структуры типов выступают в роли спецификаций, что приводит к необходимости решения задачи отображения типов на спецификации.

Обзор подходов к спецификации и верификации программ

С формальной точки зрения спецификация является логическим высказыванием, программа задается при помощи некоторого вычислительного формализма, такого как машина Тьюринга, алгоритм Маркова, рекурсивная функция, терм аппликативной вычислительной системы и т.д., верификация представляет собой процедуру проверки истинности заданного высказывания для заданной программы.

В качестве примера можно рассмотреть спецификацию и верификацию функций сортировки списка. Существует спецификация, формализующая понятие сортировки списка в виде логического высказывания. Все алгоритмы сортировки (быстрая, прямая вставка, пузырьковая и другие) должны удовлетворять этой спецификации. Однако, выполняя одну и ту же задачу, отдельные алгоритмы различаются в частных аспектах, например, в области операциональных свойств, таких как вычислительная сложность по времени, по памяти. Соответственно, эти различия могут быть формализованы дополнительными спецификациями, более специфичными, чем общая спецификация сортировки.

Из приведенного примера видно, что на спецификациях задано отношение частичного порядка, соответствующее родовидовым связям обобщения-специализации, которые будем называть вложенностью спецификаций. При этом программы можно считать спецификациями наиболее специализированного вида. Точнее, программа может сама по себе выступать в роли спецификации, и в этом случае она понимается как эталонная реализация, которой должны соответствовать верифицируемые относительно нее программы.

Для представления программ будем использовать подход *аппликативных вычислительных систем* (АВС): лямбда-исчисления [6] и комбинаторную логику [7]. С инженерной точки зрения данный подход выражается парадигмой функционального программирования – трактовкой программы как чистой математической функции. В отличие от императивного подхода, когда программа рассматривается как набор инструкций, выполнение которых ведет к изменению состояния, вычисление значения функции зависит только от входных значений и не подразумевает использование изменяемого состояния. В этом заключается основная причина выбора данного подхода: он дает возможность напрямую применять весь соответствующий математический аппарат как для синтеза программ, так и для анализа их логических свойств. Предлагаемое решение прежде всего нацелено на применение в связке с языками Scala, Coq, Agda и другими, ориентированными на использование теоретико-типового и функционального подходов, основанных на АВС. Типизированные обертки для сторонних компонентов также должны следовать функциональному стилю.

Проверка истинности, выполняемая при верификации, может быть как исчерпывающей, так и частичной, покрывающей только часть диапазона входных значений.

Полноту проверки в общем случае может гарантировать только статическая верификация, предметом которой является синтаксическая структура терма программы. Данная структура предоставляет исчерпывающую информацию о программе. Другими словами, проверка осуществляется по принципу белого ящика. В рамках АВС статическую верификацию можно свести к типизации программы [8]. Связь между типизацией и проверкой логических высказываний о программе устанавливается в теории типов через так называемый изоморфизм Карри–Говарда [9] или соответствие Брауэра–Гейтинга–Колмогорова [10, 11], которые связывают типизированные вычислительные фор-

мализмы с логическими системами: типы – с высказываниями, а термы – с предметами и доказательствами.

Полная статическая верификация на соответствие спецификации уровня логики предикатов возможна в языках программирования с зависимыми типами: Coq, Agda и т.п., но при этом нередко необходимо оснащение программы дополнительным кодом доказательств корректности. Значительно большее распространение имеют языки программирования, в которых возможности статической типизации ограничены либо отсутствуют. Это означает, что исчерпывающе может быть проверен только узкий набор свойств программ. Среди функциональных языков программирования распространены языки с полиморфными системами типов уровня системы *F*, позволяющими выражать спецификации, оперирующие типами аргументов функций, но не отдельными значениями внутри этих типов.

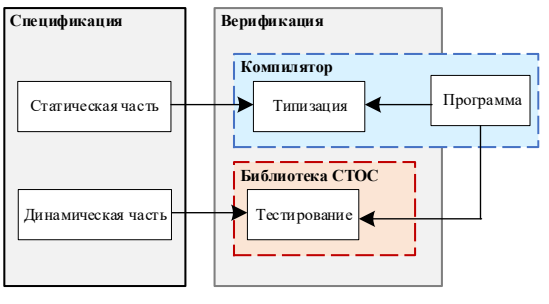
Под динамической верификацией программы понимается тестирование, то есть проверка результатов исполнения программы на тестовых данных. Поскольку осуществляется анализ внешнего поведения, а не внутренней структуры программы, можно назвать тестирование верификацией по принципу черного ящика. Построение тестовых примеров проводится так, чтобы удостовериться, что программа ведет себя в соответствии с некоторой эталонной конструкцией, которая также называется тестовым оракулом. Таким образом, можно отметить, что динамическая верификация основана на математической модели, то есть на денотационной семантике программы.

Частичность динамической верификации обусловлена тем, что в большинстве случаев за конечное время невозможно провести исчерпывающую проверку всех вариантов поведения программы из-за слишком большого или неограниченного числа вариантов входных параметров.

В рамках функционального программирования появился и затем распространился на большинство популярных языков программирования подход *случайного тестирования на основе свойств* (CTOC) [12]. Он интересен тем, что позволяет описывать тесты в виде не конкретных примеров, а свойств, имеющих форму логических высказываний. Верификация осуществляется путем генерации заданного числа случайных тестовых примеров на основе таких высказываний. Таким образом, данный подход дает возможность напрямую формулировать и проверять спецификации программ в виде логических высказываний, а полнота верификации управляется изменением числа генерируемых тестовых примеров.

Сформулируем общую схему процесса верификации, приведенную на рисунке. В зависимости от выразительности системы типов заданного языка программирования полная спецификация программы или ее часть может быть представлена соответствующими ти́повыми конструкциями и полностью верифицирована средствами статической типизации. Статическая верификация может потребовать дополнительной работы по написанию доказательств, которые будут проверены компилятором. Часть спецификации программы, не поддающаяся статической верификации, может быть представлена с использованием системы CTOC в виде свойств и протестирована динамически [13]. Динамическая верификация не требует дополнительного труда со стороны программиста, но при этом является частичной. В языках с выразительной системой типов, таких как Coq, возможно представление всей спецификации в виде ти́повой конструкции, однако по решению программиста она может быть частично верифицирована статически, а частично динамически (например, при недостатке временных ресурсов для разработки полноценных доказательств или для быстрого опровержения спецификации с целью ее дальнейшего уточнения). В языках с отсутствующими или менее выразительными системами типов динамическая часть спецификации должна формулироваться отдельно средствами библиотеки CTOC.

Распространенным способом динамической верификации сегодня является фаззинг [14]. Особенность фаззинга в том, что он не основывается на спецификации: случайные тесты проводятся таким образом, чтобы в результате итеративного процесса найти диапазоны входных



Общая схема верификации программы

General scheme of program verification

данных, на которых работает тестируемая система. С точки зрения верификации и спецификации как недостаток можно рассматривать то, что явно или неявно найденная в результате фаззинга спецификация поведения программы обычно отбрасывается и никак не фиксируется.

Сравнение подходов

Введем набор критериев для сравнения подходов к верификации:

– полнота верификации (возможность полной или только частичной верификации спецификации путем подбора контрпримера для ее опровержения);

– необходимость разработки доказательств (требуется ли разработка дополнительного кода доказательств для верификации программы);

– выразительные спецификации (возможность описания спецификаций на уровне логики предикатов);

– итеративный вывод спецификаций (возможность вывода уточненных спецификаций по результатам верификации);

– отчуждаемость спецификаций (возможность переноса спецификаций между различными языками программирования и системами тестирования).

Сравнить предлагаемое решение (QuasiType) на основе выбранных критериев с описанными выше подходами позволяет таблица.

В настоящем исследовании решается задача разработки унифицированного представления спецификаций уровня логики предикатов, оснащенного средствами импорта/экспорта для дальнейшей статической или динамической верификации. Даются общий каркас для совмещения различных подходов к верификации на основе единой спецификации, а также средства для итеративного уточнения или вывода спецификации на основе динамического тестирования.

Метод построения спецификаций

На абстрактном уровне спецификация представляет собой логическое высказывание относительно какого-либо вычислительного объекта. Алгебра спецификаций задает систему операций, при помощи которых возможно построение таких спецификаций.

Основная структура. Для построения спецификаций используем систему теории множеств и логики предикатов. Под специфицируемыми объектами будем понимать предметы, являющиеся элементами некоторого множества U, называемого универсумом. Через Set обозначим совокупность всех подмножеств U. Под $Prop$ будем понимать множество всех высказываний.

Рассмотрим основные предикаты, то есть способы построения высказываний из предметов. Классификация элементов U по принадлежности его подмножествам выполняется посредством функции $- \in -: U \times Set \to Prop$. Другим основным предикатом является предикат равенства предметов, который определяется отдельно для каждого (индекс s будет опускаться, если его значение ясно из контекста) s из Set: $- =_s -: U \times U \to Prop$.

Для построения выразительного языка спецификаций необходимо ввести в рассмотрение переменные. Множество всех переменных обозначим как Var. Переменные могут быть использованы в выражениях вместо конкретных предметов. Такие выражения будем называть формулами. Множество высказывательных формул (имеющих значение высказывания)

Общая схема верификации программы

General scheme of program verification

Критерий	Типизация уровня системы F	Типизация с зависимыми типами	CTOC	Фаззинг	QuasiType
Полнота верификации	Да	Да	Нет	Нет	Да/Нет
Не требует доказательств	Да	Нет	Да	Да	Да
Выразительные спецификации	Нет	Да	Да	Нет	Да
Итеративный вывод спецификаций	Нет	Нет	Нет	Да	Да
Отчуждаемость спецификаций	Нет	Нет	Нет	Нет	Да

обозначим как *PForm*. Множество предметных формул (имеющих значение предмета) обозначим через *UForm*. В частности, переменные можно считать простейшими предметными формулами.

Важным способом построения высказываний является утверждение о том, что некоторая формула F дает истинное высказывание при подстановке любых предметов из заданного множества S вместо некоторой переменной x. С точки зрения логики данный способ аналогичен всеобщей квантификации: $\forall x{:}\, S(F)$. Однако аналогичный смысл имеет и определенный вид суждений: $x{:}\, S \vdash F$. Следуя стилю, принятому в области алгебры и теории категорий, будем использовать второй из указанных подходов и введем операцию $(-{:}\ -\ \vdash\ -){:}\ Var \times Set \times \times PForm \to Prop$.

Заданных выше операций достаточно, чтобы определить простейшую спецификацию *eqRefl*, утверждающую рефлексивность равенства для заданного множества: $eqRefl(S{:}\, Set) \coloneqq \coloneqq x{:}\, S \vdash x{=}_S x$.

Категорная структура. Дальнейшим шагом в построении алгебры спецификаций является определение примитивов для конструирования специфицируемых предметов. В качестве основы для последующих построений примем подход теории категорий [15, 16], в частности, категории множеств *Set*. В рамках данного подхода предметами спецификаций становятся функции, которые также называют морфизмами или стрелками, а основным способом их построения – операция композиции. С формальной точки зрения это говорит о том, что введенное выше U понимается как множество морфизмов, а *Set* – как множество объектов категории. Операция композиции задается как $- \circ -{:}\, U \times U \to U$.

Необходимо заметить, что, строго говоря, композиция является частично определенной функцией, так как задана только для совместимых морфизмов, но для простоты изложения пока опустим это требование.

В рамках категории множеств объекты воспринимаются как множества и могут выступать в роли области определения (домена, *dom*) или области значений (кодомена, *cod*) морфизмов. Выделяются так называемые *Hom*-множества, содержащие морфизмы между заданными объектами: $Hom(A, B : Set) \coloneqq \{f \mid dom(f) = = A \wedge cod(F) = B\}$.

Hom-множества будут использоваться в спецификациях для ограничения множества специфицируемых морфизмов.

Композиция должна быть ассоциативна, что формулируется в виде спецификации $compAss(A, B, C, D{:}\, Set) \coloneqq f : Hom(C, D), g : Hom(B, C), h : Hom(A, B) \vdash (f \circ g) \circ h = {}_{Hom(A,D)} f \circ (g \circ h)$.

Для каждого объекта A задается так называемый единичный морфизм, или идентичность, id_A, обладающий следующими свойствами: $idL(A, B{:}\, Set) \coloneqq f{:}\, Hom(A, B) \vdash id_B \circ f = f$, $idR(A, B{:}\, Set) \coloneqq f{:}\, Hom(A, B) \vdash f \circ id_A = f$.

Несложно показать, что из указанных свойств следует однозначность идентичности для каждого объекта: пусть i и j – две идентичности для некоторого объекта, тогда $i =^{(idR)} i \circ j =^{(idL)} j$, то есть из того, что оба морфизма являются идентичностями, следует их неотличимость. Значит, уместно говорить об уникальной стрелке id_A для каждого объекта A.

Формулы и связывание. Рассмотрим подробнее использование переменных и формул. Следует заметить, что приведенные спецификации по своей синтаксической структуре противоречат определению утверждений. В частности, согласно определению, справа от символа $\vdash$ должна находиться формула из *PForm*, в то время как все приведенные спецификации содержат справа от $\vdash$ результат композиции, то есть элемент U. В свою очередь и аргументы композиции в этих спецификациях, строго говоря, не соответствуют ее области определения: f и g являются переменными, то есть элементами *Var*, в то время как композиция определена на элементах U.

Устранение указанных несоответствий состоит в проработке механизмов построения формул и связывания переменных со значениями. Это является важной задачей при построении структуры спецификаций, которая допускает не только погружение в какую-либо среду, уже оснащенную способами связывания, но и, наоборот, выгрузку спецификаций в виде самостоятельных конструкций для дальнейшего анализа.

Напомним, что формулы понимаются как конструкции, содержащие переменные. Формула приобретает значение, или означивается, в среде – некотором контексте, связывающем переменные с конкретными значениями. Введем следующие механизмы построения формул.

Во-первых, любое значение порождает формулу, цитирующую это значение, то есть принимающую это значение в любой среде: $quote{:}\, U \to UForm$.

Во-вторых, переменные понимаются как другой простейший вид формул, наоборот, полностью зависящих от среды: $Var \subseteq UForm$.

В-третьих, более сложные формулы могут получаться из более простых путем применения особых версий обычных операций, таких как композиция, и предикатов, таких как равенство, «повышенных» до уровня операций на формулах. Соответствующую операцию, преобразующую функции на U и $Prop$ в функции на $UForm$ и $PForm$, обозначим $[-]^F$. Таким образом, более точное определение, к примеру, idL имеет следующий вид: $idL(A, B: Set) :=$ $:= f: Hom(A, B) \vdash quote(id_B)[\circ]^F f[=]^F f$.

Далее будем, как и ранее, подразумевать данную структуру неявным образом.

Дополнительные структуры. Рассмотрим дополнительные структуры, используемые в спецификациях. За основу взяты стандартные определения из теории категорий, которые адаптированы к представлению в виде спецификаций на основе определенной выше структуры.

Терминальный объект. Терминальный объект будем обозначать как 1, он оснащается следующей спецификацией: $\mathrm{unit}(A: Set) :=$ $:= f, g: Hom(A, 1) \vdash f = g$.

Произведение. Декартово произведение объектов A и B обозначается как $A \times B$ и оснащается стрелками-проекциями $\pi_1: A \times B \to A$, $\pi_2: A \times B \to B$. Для каждого объекта X и пары совместимых $f: X \to A$ и $g: X \to B$ определена стрелка $\langle f, g\rangle$ таким образом, что выполняются спецификации $\mathrm{pi}_1(X, A, B: Set) := f: Hom(X, A)$, $g: Hom(X, B) \vdash \pi_1 \circ \langle f, g\rangle = f$ и $\mathrm{pi}_2(X, A, B: Set) :=$ $:= f: Hom(X, A)$, $g: Hom(X, B) \vdash \pi_2 \circ \langle f, g\rangle = g$.

Экспоненцирование. Для любых двух объектов A и B задан экспоненциальный объект B^A вместе со стрелкой $\varepsilon_{AB}: B^A \times A \to B$. Также для каждого $f: A \times B \to C$ задана стрелка $\lambda_{ABCf}:$ $: A \to C^B$. При этом должна выполняться спецификация $applyLam(A, B, C: Set) := f: A \times B \to$ $\to C \vdash \varepsilon_{BC} \circ (\lambda_{ABCf} \times id_B) = f$.

Функтор F представляет собой пару функций F_o: $Set \to Set$ и F_m: $U \to U$, таких, что справедливы спецификации $functorId\langle \mathrm{F}\rangle(A: Set) :=$ $:= F_m(id_A) = idF_{o(A)}$ и $functorComp\langle \mathrm{F}\rangle(A, B,$ $C: Set) := f: Hom(B, C), g: Hom(A, B) \vdash F_m(f \circ g) =$ $= F_m(f) \circ F_m(g)$.

Поскольку функции F_o и F_m применимы, соответственно, к объектам и стрелкам, а значит, используются в разных местах, обычно их можно отличить исходя из контекста, а индексы опустить. Исходя из того, что фактически работа ведется в одной и той же категории, все функторы являются так называемыми эндофункторами. Идентичным функтором называется функтор Id, такой, что функции Id_o и Id_m являются идентичностями.

Композиция функторов F и G представляет собой такой функтор $F \circ G$, что $(F \circ G)_o(A: Set) :=$ $:= F_o(G_o(A))$, $(F \circ G)_m(f: U) := F_m(G_m(f))$.

Композиция функторов обладает свойством ассоциативности.

Естественное преобразование из функтора F в функтор G обозначается как $\eta: F \to G$ и представляет собой функцию $\eta: Set \to U$, порождающую для каждого объекта A стрелку η_A таким образом, что справедливы спецификации $ntIn\langle\eta\rangle(A: Set) := \eta_A \in Hom(F(A), G(A))$, $ntComp\langle\eta\rangle(A, B: Set) := f: Hom(A, B) \vdash \eta_B \circ F(f) =$ $= G(f) \circ \eta_A$.

Монада состоит из эндофунктора F и двух естественных преобразований $\eta_F: Id \to F$ и $\mu: F^2 \to F$, таких, что выполняются спецификации $muComp\langle \mathrm{F}\rangle(A: Set) := \mu_F(A) \circ F(\mu_F(A)) =$ $= \mu_F(A) \circ \mu_F(F(A))$, $muEtaCompR\langle \mathrm{F}\rangle(A: Set) :=$ $:= \mu_F(A) \circ F(\eta_F(A)) = id_F(A)$, $muEtaCompL\langle \mathrm{F}\rangle(A:$ $: Set) := \mu_F(A) \circ \eta_F(F(A)) = id_F(A)$.

Таким образом, определение монады основано на вышеописанных определениях функторов и естественных преобразований, но дает мощное средство для оперирования большим количеством разнородных структур, используемых в программировании, – от списков значений до побочных эффектов и потоков ввода-вывода.

Примеры спецификаций

Рассмотрим примеры использования описанной алгебры для спецификации некоторых простых программных интерфейсов. Определение спецификаций в алгебраическом стиле позволяет решать многие задачи простым применением конкретных интерпретаторов к абстрактным спецификациям. В эту категорию попадают задачи, опирающиеся на естественную структуру спецификаций. К примеру, спецификации могут быть интерпретированы как тесты соответствующих компонентов, а также как генераторы текстовых описаний на естественном или некотором внешнем формальном языке, используемом для выгрузки спецификаций в сторонние системы для дальнейшей интерпретации, формальной верификации и т.п.

Спецификация изменяемых значений. Приведем пример простейшей спецификации стандартного интерфейса для работы с изменяемым значением – так называемых аксессоров `get` и `set`. Этот довольно простой пример в то же время является достаточно общим, практически напрямую применяемым к другим задачам чтения и изменения данных в различных про-

граммных интерфейсах, таких как работа с БД, операции REST API и другие.

Предложим следующие спецификации: *get* := := *get* ∈ *Hom*(1, ℕ), *set* := *set* ∈ *Hom*(ℕ, 1), *setGet* := *get* ∘ *set* = *id*ℕ.

Спецификации *get* и *set* задают типы соответствующих функций, в то время как спецификация *setGet* характеризует смысловую связь этих операций.

Недостатком рассмотренной спецификации является то, что она не учитывает побочные эффекты, которые неизбежно связаны с обработкой изменяемых значений. Вследствие этого при интерпретации данной спецификации в виде тестов при их параллельном выполнении тест *setGet* в большинстве случаев возвращает ошибку.

Более точно это несоответствие выражается нарушением спецификации вида *pureness* := := $i : Hom(1, \mathbb{N}) \vdash get = get \circ set \circ i$, справедливость которой требуется исходя из свойств терминального объекта 1 и тождественного морфизма. Поскольку из любого объекта существует только единственный морфизм, любой морфизм типа $1 \to 1$ должен совпадать с тождественным морфизмом id_1. Одним из таких морфизмов является $set \circ i$ для любого $i \in$ $\in Hom(1, \mathbb{N})$. Но из того, что $set \circ i = id_1$, следует, что $get \circ set \circ i = get \circ id_1 = get$. Очевидно, что это равенство нарушается в зависимости от того, изменяется ли начальное значение при выполнении операции *set*. В совокупности вышесказанное означает, что неверно использовать 1 в роли области значений для *set*. Содержательно при трактовке 1 как одноэлементного множества такое определение говорило бы о том, что результат выполнения *set* всегда одинаков, однако это не так: выполнение *set* с различными аргументами дает различные состояния, хотя содержание этих состояний не возвращается в явном виде. Аналогично неверно рассматривать 1 и как значение *get*, так как различные начальные состояния могут приводить к разным результатам выполнения *get*.

Учет побочных эффектов. Исправить приведенную выше спецификацию можно, введя в рассмотрение побочные эффекты вычислений, представленные через соответствующую монаду. Пусть *IO* – это функтор и оснащающая его монада, соответствующая реализации побочных эффектов. Рассмотрим спецификацию аксессоров, учитывающую связь операций *get* и *set* с побочными эффектами: *getPure* := *get* ∈ $\in Hom(1, IO(\mathbb{N}))$, *setPure* := *set* ∈ $Hom(\mathbb{N}, IO(1))$, *setGetPure* := $\mu_{IO(\mathbb{N})} \circ IO(\mathbb{N}) \circ set = \eta_{IO(\mathbb{N})}$.

Данная спецификация разрешает несоответствие области значений аксессоров терминальному объекту. Содержательно замена 1 на *IO*(1) явно описывает то, что результат этих операций не предопределен, а скрыто зависит от внешнего состояния или же изменяет его. Соответственно, в отличие от 1 объект *IO*(1) не обладает свойствами, которые приводили к противоречиям. В то же время монадическая структура *IO* позволяет проводить необходимые композиции операций. С точки зрения интерпретации спецификаций как тестов обновленные спецификации сами по себе не делают успешными тесты, ранее приводящие к ошибкам. Например, параллельное выполнение тестов все еще может завершаться неудачей. Однако обновленные спецификации в явном виде сообщают внешней среде о наличии побочных эффектов, и, имея эти данные, среда выполнения тестов может принимать решения о том, какие дополнительные свойства эффектов проверяются. Конечно, возможно составление дополнительных спецификаций поведения операций при параллельном выполнении.

Язык спецификаций

Представленная алгебраическая структура спецификаций хорошо подходит для интерпретации спецификаций в разных целях. Однако интерпретация возможна, если существует закодированная структура конкретной спецификации. Если определение спецификации отсутствует и стоит задача считывания ее в виде данных или же, наоборот, сохранения закодированной спецификации в виде данных для дальнейшего считывания, то требуется язык спецификаций, на котором можно записать такие данные. Являясь двойственными представлениями одной и той же сущности, язык и алгебра спецификаций строятся похожим образом и могут быть выведены друг из друга.

Наиболее простым способом построения языка спецификаций является ввод всех описанных выше конструкций в виде отдельных синтаксических единиц.

Начнем с базовых возможностей, позволяющих описывать высказывания (*P*) относительно предметов (*U*) и множеств (*S*), а также строить высказывательные формулы (*PF*):

$P ::= U \in S \mid U = {}_S U \mid PF^{\downarrow}$,

$PF ::= S \vdash PF$.

Далее в грамматику вводятся средства определения предметов и множеств, основанные на категорном подходе, необходимые для форму-

лировки спецификаций через композиции функций (морфизмов), а также с учетом областей определения и значений:

$U ::= U \circ U \mid id_S$,

S ::= $Hom(S, S)$.

Полученную грамматику можно расширять, добавляя новые конструкции. Приведем пример добавления терминального объекта и произведений:

$U ::= \ldots \mid \langle U, U \rangle \mid \pi_1 \mid \pi_2$,

$S ::= \ldots \mid S \times S \mid 1$.

Отдельная синтаксическая категория формул *PF* введена как представление результата квантификации $- \vdash -$ и требует наличия методов представления связывания, описанных выше. Введем их следующим образом:

$UF ::= 'U' \mid \overline{n} \mid UF \circ UF \mid \langle UF, UF \rangle$,

$PF ::= \ldots \mid UF \in S \mid UF = {}_S UF$.

Здесь *UF* представляет собой синтаксическую категорию предметных формул, которые могут содержать связанные переменные. При этом связывание осуществляется не по именам связанных переменных, а по соответствующим им индексам де Брейна [17]: *n*-й индекс обозначается как $\overline{n}$. Формула, цитирующая некоторый предмет *u*, обозначена как $'u'$. Кроме того, расширение грамматики содержит повышенные до уровня формул аналоги операций $- \circ -$, $\langle -, - \rangle$, $- \in -$ и $- = -$, которые используют тот же символ, что и оригинальная операция, но отличаются друг от друга окружающим контекстом.

Интерпретация

Представленную алгебру спецификаций можно интерпретировать для разных целей. Технически используется подход Tagless Final [18, 19]. В рамках объектно-ориентированного подхода Scala это означает, что основная алгебра реализуется в виде системы абстрактных классов и трейтов, а интерпретаторы – в виде конкретных реализаций этих абстрактных определений. На формальном уровне интерпретация понимается в смысле, близком к понятию интерпретации формальной системы: всем носителям абстрактной алгебраической системы ставятся в соответствие конкретные множества или типы программных объектов, а всем операциям – конкретные операции на соответствующих типах.

Интерпретация для тестирования. Целью данного проекта является построение системы описания спецификаций, погружаемой в различные вычислительные среды для верификации конкретных программных компонентов. Соответственно, основной формой интерпретации спецификаций является генерация тестов в рамках подхода СТОС.

Настоящий проект реализуется на языке Scala, в связи с чем спецификации интерпретируются в виде определений свойств ScalaCheck. Фактически разработанная структура спецификаций во многом повторяет структуру свойств `Prop` системы ScalaCheck и основана на ней. В то же время спецификации QuasiType являются более высокоуровневыми, что позволяет проводить их выгрузку для последующего погружения в другие среды – в реализации подхода СТОС на других языках. С этой точки зрения алгебра QuasiType предоставляет общую структуру спецификаций, относительно которой QuickCheck, ScalaCheck и другие играют роль реализаций в определенных вычислительных средах, а соответствующие интерпретаторы QuasiType устанавливают соответствие между абстрактной алгеброй и конкретной реализацией.

Основными составными частями систем СТОС являются структуры для определения свойств, как правило, имеющих тип *`Prop`*, и средства генерации случайных значений различных типов, обычно сводящихся к структурам типов `Gen` (генераторы) и `Cogen` (когенераторы).

Например, свойство ассоциативности композиции в ScalaCheck выглядит как

```
   forAll((f: Int => Int, g: Int =>
=> Int, h: Int => Int, x: Int) =>
     f.compose(g).compose(h)(x) ==
==f.compose(g.compose(h))(x)):
Prop,
```

и его тестирование состоит в выполнении содержащейся в нем функции для набора случайных аргументов. Для генерации случайных аргументов типа *T* используется генератор типа Gen[*T*], и он передается в forAll в качестве неявного параметра. Для обеспечения повторяемости тестов генерация псевдослучайных значений управляется особым значением типа Seed. Таким образом, Gen[*T*] фактически представляет собой функцию Seed => *T*. Заметим, что для проверки ассоциативности композиции требуется генерация случайных значений функционального типа Int => Int. Случайная функция, с одной стороны, должна возвращать случайный результат, но, с другой, не должна однозначно зависеть от аргумента функции. Таким образом, для генерации случайной функции типа *A* => *B* необходим способ случайного значения типа *B*, в то же время управ-

ляемого значением типа *A*. Исходя из того, что все случайные значения управляются значением Seed, данная задача сводится к наличию способа получения Seed из *A* – то есть функции типа *A* => Seed, который также именуется Cogen[*T*]. Таким образом, эти когенераторы необходимы для построения генераторов функциональных значений с соответствующей областью определения. Поскольку поддерживаются спецификации функций с побочными эффектами, выполненная реализация использует библиотеку scalacheck-effect, дополняющую ScalaCheck возможностями использования побочных эффектов на основе библиотек cats и cats-effect путем подмены обычных свойств типа *Prop* свойствами с эффектом *PropF*[*TEff*].

Интерпретация основывается на следующих носителях:

– высказываниям *Prop* ставятся в соответствие значения типа *PropF*[*IO*];

– морфизмам из *U* ставится в соответствие тип функций Any => Any, позволяющий избегать ошибок типизации, опираясь на соответствующие динамические проверки;

– множествам *Set* ставятся в соответствие структуры, позволяющие, с одной стороны, генерировать значения соответствующего типа для выполнения тестов, а с другой, проверять истинность базовых предикатов.

Класс *Set*, интерпретирующий множества из *Set*, представляет собой класс кортежей вида

```
( gen: Gen[Any],
  cogen: Cogen[Any],
  elem(m: Any): PropF[IO],
  eq(x: Any, y: Any):
PropF[IO]).
```

Здесь gen и cogen представляют соответственно генератор и когенератор. Текущая реализация интерпретатора является нетипизированной в том смысле, что не использует или, точнее, не сохраняет конкретные типы, заменяя их общим типом Any: каждое множество соотносится с конкретным генератором значений конкретного типа, однако затем этот тип стирается, не влияя на работу созданного генератора. Значения elem и eq интерпретируют, соответственно, предикаты $-\in_s-$ и $-=_s-$ для соответствующего множества *s*.

Помимо структуры *Set*, остальная часть интерпретатора практически напрямую ставит в соответствие операциям на морфизмах соответствующие им операции на функциях Scala, а также интерпретирует остальные возможности методами стандартной библиотеки, библиотек cats и других. Например, произведение интерпретируется стандартным классом пар и оснащающими его функциями проекций. Монада, определяемая некоторым функтором и операциями η и μ, интерпретируется монадой из cats и соответствующими операциями fmap, pure и flatten.

После применения интерпретатора спецификации, описывающие поведение операций η, μ и других, автоматически становятся исполняемыми тестами соответствующих методов функций pure и flatten и т.д.

Интерпретация для транслирования. Возможность выгрузки и переноса спецификаций между различными средами или языками программирования (транслирование спецификаций) достигается путем представления их в виде не абстрактных выражений, а конкретных значений. Структура таких значений соответствует грамматике языка спецификаций, приведенной ранее.

Интерпретатор для целей транслирования состоит из иерархии классов, представляющих собой синтаксические сущности грамматики, и реализации абстрактной алгебры, ставящей в соответствие каждой операции конструктор соответствующего класса. Такой интерпретатор используется для экспорта спецификаций в виде конкретных структур, которые затем могут быть сериализованы с использованием стандартных коммуникативных форматов, таких как JSON, YAML, XML и другие.

Помимо интерпретатора, позволяющего выгружать спецификации, целесообразно реализовать анализатор для считывания спецификаций из внешних источников. Часть анализатора, отвечающая за чтение или десериализацию из внешних форматов, может быть автоматически сгенерирована стандартными средствами на основе структуры классов грамматики языка. Остается реализовать парсер, переводящий полученные значения в соответствующие им выражения в терминах алгебры спецификаций. Полученные выражения затем могут быть интерпретированы другими интерпретаторами для тех или иных целей.

Алгебра и язык спецификаций изоморфны друг другу. Значит, импорт любой спецификации с ее последующим экспортом должен соответствовать тождественному преобразованию, то есть давать в качестве результата исходную спецификацию. Выполненные в составе системы QuasiType реализации интерпретатора и парсера оснащены соответствующей спецификацией, проверяющей указанную изоморфность.

Интерпретация для статической верификации. Рассмотрим, каким образом предложенное представление спецификации может быть использовано не для динамической, а статической верификации программного кода при помощи типизации. Подход к статической проверке состоит в построении интерпретатора абстрактных спецификаций, связывающего используемые в спецификациях множества с типами целевого языка. Возможности верификации такого интерпретатора ограничиваются возможностями, предоставляемыми соответствующей системой типов: например, поддерживается ли типизация функциональных значений. Если же язык поддерживает в том числе и зависимые типы, то подобный интерпретатор может связывать с типовыми выражениями всю спецификацию, включая используемые в ней предикаты, и дальнейшая верификация может быть целиком осуществлена статически в форме написания доказательств. В свою очередь, верифицируемый код встраивается в спецификацию и таким образом должен проходить статическую проверку соответствующих типов при компиляции.

Следует отметить, что интерпретаторы для тестирования и для статической проверки являются граничными случаями и востребованными могут оказаться гибридные интерпретаторы, покрывающие случайными тестами те части спецификации, которые не выразить в рамках системы типов целевого языка программирования.

Обсуждение: применение подхода к уточнению спецификаций

Возможности декомпозиции, преобразования и повторной интерпретации спецификаций позволяют строить правила и алгоритмы их уточнения и совершенствования.

В качестве примера рассмотрим задачу выявления в спецификациях программных интерфейсов функций-аксессоров для работы с изменяемыми значениями с последующим уточнением этих спецификаций. Поскольку преобразование основано в том числе и на информации об именах функций, будем считать, что задана функция $name\colon U \to String$, возвращающая имя заданного морфизма. Алгоритм данного преобразования можно представить в виде следующего правила вывода:

$$name(g) = \texttt{"get⟨n⟩"} \quad g \in Hom(\mathbb{N}, 1),$$
$$name(s) = \texttt{"set⟨n⟩"} \quad s \in Hom(1, \mathbb{N})$$
$$\Rightarrow \quad g \circ s = id_{\mathbb{N}},$$

где n является изменяемым значением, доступ к которому предоставляют g и s.

Данное правило выводит спецификацию согласованности «геттера» и «сеттера». Приведена версия, не учитывающая побочные эффекты. Идея состоит в том, чтобы на основе анализа имен и типов, например, методов некоторого класса, делать вывод о возможности воспринимать их в качестве аксессоров для некоторых значений и затем проверять справедливость этого вывода путем запуска уточненной спецификации в виде тестов.

Заключение

В настоящей работе представлен подход к спецификации программных компонентов, дающий отчуждаемые спецификации, оснащенные средствами погружения в различные вычислительные среды для дальнейшей динамической или статической верификации. Он является основой для построения экосистемы программных средств с целью верификации и анализа функционала разнородных компонентов. Подход реализуется в проекте QuasiType (https://www.tylip.com/ru/quasitype), выполняемом на языке Scala.

Перспективным видится дальнейшее развитие подхода в различных направлениях. В частности, планируются разработка реализаций на других языках программирования, а также интеграция с такими системами, как Coq и Agda, позволяющими осуществлять логический вывод производных спецификаций. Планируется применение разработанного подхода для автоматизации верификации типизированных оберток (фасадов) над нетипизированными компонентами, такими как библиотеки JavaScript, средства работы с контейнеризированными приложениями и их API. Ограничением к применению подхода на данный момент является требование к наличию первичного фасада, на основе которого будет построена спецификация, способная затем итеративно верифицироваться и уточняться. Однако данное ограничение может ослабляться путем разработки анализаторов программного кода, автоматизирующих построение начальных спецификаций.

Список литературы

1. Shapkin P. Automation of configuration, initialization and deployment of applications based on an algebraic approach. Procedia Comput. Sci., 2022, vol. 213, pp. 785–792. doi: 10.1016/j.procs.2022.11.135.

2. Pierce B.C., de Amorim A.A., Casinghino Ch., Gaboardi M. et al. Logical foundations. Software Foundations Ser., 2018, vol. 1. URL: https://www.seas.upenn.edu/~cis5000/current/sf/lf-current/index.html (дата обращения: 11.05.2024).

3. Methni A., Lemerre M., Hedia B.B. et al. An approach for verifying concurrent C programs. Proc. 8th Junior Researcher Workshop on Real-Time Computing, 2014, pp. 33–36.

4. Lau H., Nestmann U. Java goes TLA+. Proc. V Int. Conf. on Theoretical Aspects of Software Eng., 2011, pp. 117–124. doi: 10.1109/TASE.2011.44.

5. Filliâtre J.-C., Pascutto C. Ortac: Runtime assertion checking for OCaml (tool paper). In: LNPSE. Proc. RV, 2021, vol. 12974, pp. 244–253. doi: 10.1007/978-3-030-88494-9_13.

6. Барендрегт Х. Ламбда-исчисление. Его синтаксис и семантика. М.: Мир, 1985. 606 с.

7. Вольфенгаген В.Э. Комбинаторная логика в программировании. Вычисления с объектами в примерах и задачах. М.: изд-во ЮрИнфоР МГУ, 2008. 204 с.

8. Barendregt H. Introduction to generalized type systems. J. of Functional Programming, 1991, vol. 1, no. 2, pp. 125–154. doi: 10.1017/S0956796800020025.

9. Sørensen M.H., Urzyczyn P. Lectures on the Curry-Howard isomorphism. In: Studies in Logic and the Foundations of Mathematics, 2006, vol. 149, pp. 1–442.

10. Heyting A. Die intuitionistische Grundlegung der Mathematik. Erkenntnis, 1931, vol. 2, pp. 106–115. doi: 10.1007/BF02028143.

11. Колмогоров А.Н. Избранные труды. Математика и механика. М.: Наука, 1985. 476 с.

12. Claessen K., Hughes J. QuickCheck: A lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices, 2011, vol. 64, no. 4, pp. 53–64. doi: 10.1145/1988042.1988046.

13. Dénès M. Hritcu C., Lampropoulos L., Paraskevopoulou Z., Pierce B.C. QuickChick: Property-based testing for Coq. Proc. the Coq Workshop, 2014, vol. 125, 126 p.

14. Морозов А.В., Панамарев Г.Е. Исследование технологий повышения доверия к специальному программному обеспечению с применением инструментальных средств, реализующих фаззинг-тестирование // Изв. РАРАН. 2022. № 3. С. 129–136.

15. Маклейн С. Категории для работающего математика; [пер. с англ.]. М.: Физматлит, 2004. 352 с.

16. Pierce B.C. Basic category theory for computer scientists. The Mit Press Publ., 1991, 113 p.

17. de Bruijn N.G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Proc. Indagationes Mathematicae, 1972, vol. 75, no. 5, pp. 381–392.

18. Kiselyov O. Typed tagless final interpreters. In: LNTCS. Generic and Indexed Programming, 2012, vol. 7470, pp. 130–174. doi: 10.1007/978-3-642-32202-0_3.

19. Carette J., Kiselyov O., Shan Ch.-Ch. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. of Functional Programming, 2009, vol. 19, no. 5, pp. 509–543. doi: 10.1017/S0956796809007205.

Software & Systems doi: 10.15827/0236-235X.149.065-076 2025, 38(1), pp. 65–76

A system of verifiable software component specifications with embedding and extraction

Pavel A. Shapkin [1] ✉

[1] National Research Nuclear University “MEPhI”,
Moscow, 115409, Russian Federation

For citation
Shapkin, P.A. (2025) ‘A system of verifiable software component specifications with embedding and extraction’, *Software & Systems*, 38(1), pp. 65–76 (in Russ.). doi: 10.15827/0236-235X.149.065-076
Article info
Received: 06.03.2024 After revision: 12.07.2024 Accepted: 24.07.2024

Abstract. This paper focuses on specification and verification of software systems and their components. It researches a unified specification language that correlates with both random testing systems and static verification tools based on type systems. A variety of programming languages, configuration systems, deployment and other tools require developers to make efforts to integrate them. Verifiable component specifications help to simplify the task. The paper proposes an approach to a unified specification representation integrated with systems for both static type checking and dynamic testing. This solution relies on methods of applicative computing and type theory. It is a conceptual framework for building specifications embedded in various software environments. The lack of static verification capabilities due to limited type systems is eliminated by dynamic testing to some extent. The author implements testing by interpreting specifications into definitions for property-based random testing systems. The practical significance of the proposed approach is automation of the process of constructing typed wrappers, or facades, which are essential for using components from less typed environments

in programming languages with more expressive type systems. The approach automates both the verification of such wrappers and the methods of their construction by defining specification refinement operations. In practice, this allows detecting errors in typing of third-party components at early development stages. The paper gives examples of program specifications with side effects. A basis for specifications is category theory formalizations. The author also analyzes approaches to translating specifications into other representations and to iteratively improving specifications by transforming them.
Keywords: software systems, verifiable, verification, semantics, logic, typing

References

1. Shapkin, P. (2022) 'Automation of configuration, initialization and deployment of applications based on an algebraic approach', *Procedia Comput. Sci*., 213, pp. 785–792. doi: 10.1016/j.procs.2022.11.135.

2. Pierce, B.C., de Amorim, A.A., Casinghino, Ch., Gaboardi, M. et al. (2018) 'Logical foundations', *Software Foundations Ser*., 1, available at: https://www.seas.upenn.edu/~cis5000/current/sf/lf-current/index.html (accessed May 11, 2024).

3. Methni, A., Lemerre, M., Hedia, B.B. et al. (2014) 'An approach for verifying concurrent C programs', *Proc. 8th Junior Researcher Workshop on Real-Time Computing*, pp. 33–36.

4. Lau, H., Nestmann, U. (2011) 'Java goes TLA+', *Proc. V Int. Conf. on Theoretical Aspects of Software Eng.*, pp. 117–124. doi: 10.1109/TASE.2011.44.

5. Filliâtre, J.-C., Pascutto, C. (2021) 'Ortac: Runtime assertion checking for OCaml (tool paper)', in *LNPSE. Proc. RV*, 12974, pp. 244–253. doi: 10.1007/978-3-030-88494-9_13.

6. Barendregt, H.P. (1981) *The Lambda Calculus. Its Syntax and Semantics*. NY: North-Holland Publ. Company, 654 p. (Russ. ed.: (1985) Moscow, 606 p.).

7. Wolfengagen, V.E. (2003) *Combinatory Logic in Programming.* 347 p. (Rus. ed.: (2008) Moscow, 204 p.).

8. Barendregt, H. (1991) 'Introduction to generalized type systems', *J. of Functional Programming*, 1(2), pp. 125–154. doi: 10.1017/S0956796800020025.

9. Sørensen, M.H., Urzyczyn, P. (2006) 'Lectures on the Curry-Howard isomorphism', in *Studies in Logic and the Foundations of Mathematics*, 149, pp. 1–442.

10. Heyting, A. (1931) 'Die intuitionistische Grundlegung der Mathematik', *Erkenntnis*, 2, pp. 106–115. doi: 10.1007/BF02028143.

11. Kolmogorov, A.N. (1985) *Selected Proc. Mathematics and Mechanics*. Moscow, 476 p. (in Russ.).

12. Claessen, K., Hughes, J. (2011) 'QuickCheck: A lightweight tool for random testing of Haskell programs', *ACM SIGPLAN Notices*, 64(4), pp. 53–64. doi: 10.1145/1988042.1988046.

13. Dénès, M. Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., Pierce, B.C. (2014) 'QuickChick: Property-based testing for Coq', *Proc. the Coq Workshop*, 125, 126 p.

14. Morozov, A.V., Panamarev, G.E. (2022) 'Research of technologies for increasing confidence in special software using tools that implement fuzzing testing', *Proc. RARAS*, (3), pp. 129–136 (in Russ.).

15. Mac Lane, S. (1998) *Categories for the Working Mathematician*. Springer Publ., 315 p. (Russ. ed.: (2004) Moscow, 352 p.).

16. Pierce, B.C. (1991) *Basic Category Theory for Computer Scientists*. The Mit Press Publ., 113 p.

17. de Bruijn, N.G. (1972) 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem', *Proc. Indagationes Mathematicae*, 75(5), pp. 381–392.

18. Kiselyov, O. (2012) 'Typed tagless final interpreter', in *LNTCS. Generic and Indexed Programming*, 7470, pp. 130–174. doi: 10.1007/978-3-642-32202-0_3.

19. Carette, J., Kiselyov, O., Shan, Ch.-Ch. (2009) 'Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages', *J. of Functional Programming*, 19(5), pp. 509–543. doi: 10.1017/S0956796809007205.

Авторы
Шапкин Павел Александрович [1],
к.т.н., доцент, pashapkin@mephi.ru

[1] Национальный исследовательский ядерный университет «МИФИ», г. Москва, 115409, Россия

Authors
Pavel A. Shapkin [1], Cand. of Sci. (Engineering), Associate Professor, pashapkin@mephi.ru

[1] National Research Nuclear University "MEPhI", Moscow, 115409, Russian Federation