Проблемы логики и методологии науки

УДК: 159.955.3+004.42
DOI: 10.15372/PS20260101
EDN: WEBRDA

А.И. Разумовский

АБСТРАГИРОВАНИЕ И ЗАКОНОМЕРНОСТИ В РАЗРАБОТКЕ АЛГОРИТМА

В статье исследуется роль абстрагирования в информатике как инструмента управления сложностью через иерархию уровней абстракции, где каждый уровень инкапсулирует предыдущий, формируя логические конструкции. Объектно ориентированное программирование демонстрирует, как абстракции организуют взаимодействие сущностей, отражая философскую концепцию информации как основы реальности. В отличие от естественных наук, информатика конструирует свой предмет через абстракции – алгоритмы, структуры данных, устанавливая «закономерности» в виде инвариантов, которые, подобно природным законам, обеспечивают предсказуемость, но допускают временное нарушение для адаптации. Баланс между свободой проектирования и необходимостью порядка подчеркивает уникальность дисциплины как своего рода метанауки, сочетающей творчество и формальные ограничения.

Ключевые слова: абстрагирование; уровни абстракции; объектно-ориентированное программирование; информатика; инварианты

A.I. Razumowsky

ABSTRACTING AND PATTERNS IN ALGORITHM DESIGN

The article studies the role of abstracting in computer science as a tool for managing complexity through a hierarchy of abstraction levels, where each level encapsulates the previous one, forming logical constructs. Object-oriented pro-

gramming demonstrates how abstractions organize the interaction of entities, reflecting the philosophical concept of information as the foundation of reality. Unlike the natural sciences, computer science constructs its subject matter through abstractions – algorithms and data structures – establishing “patterns” in the form of invariants that, just like natural laws, ensure predictability but allow for temporary violations for adaptation. The balance between the freedom of design and the necessity of order underscores the uniqueness of the discipline as a kind of meta-science, which combines creativity and formal constraints.

Keywords: abstracting; levels of abstraction; object-oriented programming; computer science invariants

Для глубокого понимания роли абстракции в разработке алгоритмов необходимо погрузиться в среду, в которой функционируют специалисты по информатике, – мир, где доминируют абстракции. Метод «уровней абстракции» признается ключевым подходом в философских исследованиях компьютерных наук, формируя основу для анализа сложных систем через иерархию упрощенных моделей [5]. Эта методология тесно связана с задачей выявления закономерностей, которые определяют структуру, логику и предпосылки алгоритмических решений. Абстрагирование, таким образом, выступает не только инструментом упрощения, но и механизмом обнаружения универсальных принципов, лежащих в основе вычислительных процессов.

В контексте информационного описания реальности Л. Флориди [5] обращается к практике разработки программного обеспечения, выделяя объектно ориентированное программирование (ООП) как яркий пример абстракции в действии. ООП, с его акцентом на инкапсуляции, наследовании и полиморфизме, демонстрирует, как абстракции позволяют управлять сложностью, разбивая систему на уровни взаимодействующих сущностей. Этот подход отражает философский взгляд Л. Флориди на информацию как фундаментальный компонент реальности, где абстракции служат мостом между теоретическими моделями и их практической реализацией.

Углубляя понимание абстракции и ее роли в установлении паттернов, важно оценить, как эти концепции влияют на более широкие философские тезисы Л. Флориди. Например, его идея «информационной онтологии» [4], где реальность интерпретируется через призму данных и структур, перекликается с тем, как абстракции в компью-

терных науках организуют хаотичные элементы в предсказуемые системы. Исследование этой взаимосвязи позволяет не только уточнить методологию алгоритмического проектирования, но и переосмыслить философские рамки, в которых анализируется роль информации в современном мире.

Информатика, в отличие от естественных наук, не открывает законы существующей реальности, а создает собственный предмет изучения через многослойные абстракции. Ее базовые элементы – программы, алгоритмы, структуры данных – не подчиняются физическим ограничениям, а существуют в пространстве логических конструкций, где правила определяются не природой, а проектировщиком. Это превращает дисциплину в своего рода «метанауку»: ее законы не только описывают поведение объектов, но и предписывают, каким оно должно быть. Например, принцип инкапсуляции в ООП или условия корректности алгоритма – это не наблюдаемые закономерности, а сознательно установленные рамки, которые формируют «архитектуру» абстрактных вселенных вычислений. Однако ключевая особенность таких законов – их условность.

Программист может переопределять правила: ослабить требование к балансировке красно-черного дерева ради упрощения кода или пожертвовать строгой типизацией для гибкости, создавая новые паттерны. Но эта свобода не абсолютна. Абстракции информатики всегда конечны: их логические миры проецируются на физические машины, где нарушение внутренних инвариантов (например, утечка памяти или коллизии хешей) приводит к сбоям, замедлениям, ошибкам. Даже виртуальная «бесконечность» вычислительных ресурсов в теории автоматов сталкивается с конечностью процессорных тактов и объема памяти. Поэтому законы информатики балансируют между двумя полюсами: с одной стороны, они – творческий инструмент, позволяющий конструировать реальности с заданными свойствами (как в случае с машиной Тьюринга или lambda-исчислением), с другой – дисциплинарный каркас, который не дает этим реальностям рассыпаться под грузом противоречий. Именно эта двойственность превращает написание кода не только в инженерную задачу, но и в акт философского выбора: какие ограничения принять как данность, а какие – пересмотреть, чтобы абстракция оставалась одно-

временно и управляемой, и функциональной. Даже такие фундаментальные концепции, как сложность алгоритмов (*O*-нотация) или принципы распределенных систем (*CAP*-теорема), по сути, являются договоренностями – формальными «правилами игры», которые работают до тех пор, пока сообщество признает их полезными. В этом суть уникальности информатики: ее законы не высечены в камне, а существуют как динамичный диалог между свободой моделирования и необходимостью предсказуемости, между полетом мысли программиста и суровой конкретикой реальности.

Информатика занимает уникальное положение в системе научного знания, поскольку конструирует собственный предмет исследования, в отличие от естественных наук, изучающих объективно существующие природные явления, или социальных наук, анализирующих закономерности человеческого поведения. Если естественные и социальные науки описывают и объясняют реальность через наблюдение и интерпретацию, то информатика оперирует искусственно созданными абстракциями – процедурами, типами данных, активными объектами и виртуальными машинами, которые одновременно выступают и как инструмент анализа, и как самостоятельный объект изучения. Эти абстракции не имеют аналогов в физическом мире, что принципиально отличает их от предметов традиционных научных дисциплин.

Фундаментальное отличие информатики от математики, как отмечают С. Маклейн [12] и Р. Милнер [13], заключается в целевом фокусе: если математика создает структуры для логического вывода, то информатика проектирует динамические модели взаимодействия. Хотя оба направления сходятся в использовании формальных методов (например, при доказательстве теорем о формальных языках), ключевой задачей информатики остается разработка программного обеспечения. Этот процесс предполагает многоуровневое моделирование взаимодействий, где каждый уровень абстракции определяет специфику отношений между компонентами системы. На низшем уровне программное обеспечение описывает манипуляции с ячейками памяти и процессорными инструкциями; на промежуточном – алгоритмически регулирует взаимодействие подпрограмм; на выс-

шем – реализует коммуникацию вычислительных процессов или интерфейсы «пользователь – система».

Понятие уровня абстракции раскрывается через историческую эволюцию программирования: прогресс в разработке ПО сопровождался последовательным дистанцированием от машинно ориентированных сущностей (инструкций, типов данных) в пользу концепций, оперирующих логическими паттернами Л. Флориди [5]. Языковая абстракция, воплощенная в высокоуровневых конструкциях (классы, функции, полиморфизм), выполняет роль семиотического посредника, скрывающего аппаратные детали за синтаксическими структурами. Это позволяет программистам концентрироваться на проектировании взаимодействий, делегируя трансляцию абстракций в машинный код компиляторам и интерпретаторам. Таким образом, управление сложностью современных программных систем достигается за счет иерархии уровней абстракций, где каждый последующий уровень инкапсулирует предыдущий, фокусируясь на определенном аспекте вычислительного процесса.

На фундаментальном физическом уровне вычислительный процесс представляет собой последовательность изменений электронных состояний машины, где каждое состояние определяется распределением электрических зарядов в элементах памяти и процессора. Однако программистская деятельность принципиально дистанцирована от манипуляций с этими физическими состояниями благодаря использованию языков программирования, оперирующих абстракциями более высокого порядка.

Программист, работающий на ассемблере, абстрагируется от логических схем и напряжений, оперируя регистрами, ячейками памяти и подпрограммами. Переход к языкам вроде *C* позволяет игнорировать ассемблерные конструкции, заменяя их переменными, указателями, структурами и функциями. На следующем уровне абстракции, как в случае с *Java*, разработчик отказывается от ручного управления памятью и низкоуровневых операций в пользу объектов и методов. Каждый языковой слой вводит не просто новые термины, но принципиально иные концептуальные модели, расширяющие выразительные возможности кода. Например, переход от *C* к *Java* предоставляет доступ к активным объектам – структурам данных, инкап-

сулирующим поведение и имитирующим взаимодействие распределенных программных агентов, одновременно автоматизируя рутинные операции вроде управления памятью.

Повышение уровня абстракции языка напрямую коррелирует с его экспрессивностью: разработчики получают возможность оперировать концептами, непосредственно соответствующими предметным сущностям моделируемой области. Эта выразительность достигается за счет маскировки сложности нижележащих уровней взаимодействия – от машинных инструкций до алгоритмических паттернов. Историческая эволюция языков программирования отражает данный процесс: переход от ассемблера к языкам высокого уровня стал возможен благодаря развитию компиляторов, усложнению системы времени выполнения, а также росту вычислительной мощности и объема памяти. Каждый новый этап абстракции не устраняет предыдущие уровни, но переопределяет их как неявный фундамент, позволяя фокусироваться на проектировании функциональности, а не на технической реализации.

В компьютерных науках, помимо языковой абстракции, существуют иные формы абстрагирования, такие как процедурная абстракция и абстракция данных. Их объединяет принцип сокрытия информации – метод, при котором детали реализации маскируются между уровнями описания системы. Этот подход находит параллели в философской концепции уровней абстракции (УА), предложенной Л. Флориди. Согласно его методу, УА служат инструментом для разрешения концептуальных противоречий в философии за счет структурирования анализа: явного формулирования допущений, обеспечения сравнимости подходов и повышения точности терминологии [5]. Хотя такой метод эффективно устраняет «понятийную путаницу», вопрос о степени соответствия между философскими УА Л. Флориди и их аналогами в компьютерных науках требует детализации.

Л. Флориди определяет уровень абстракции как вектор типизированных переменных, доступных для наблюдения или интерпретации, при этом ключевым аспектом становится намеренное игнорирование определенных свойств системы. Например, УА для покупателя вина может включать переменные «производитель», «цена»,

«регион», тогда как УА дегустатора фокусируется на «кислотности», «аромате», «послевкусии» [5, p. 309]. Таким образом, выбор УА определяется прагматической целью: он отражает не объективную структуру реальности, но оптимизированную перспективу для решения конкретной задачи.

В контексте программирования уровни абстракции также предоставляют различные наблюдаемые объекты для описания вычислительных процессов. Однако их выбор обусловлен не только целеполаганием, но и необходимостью баланса между выразительностью и сложностью. Так, ассемблер требует работы с регистрами и подпрограммами, язык *C* – с переменными и указателями, *Java* – с объектами и методами. Каждый следующий уровень скрывает архитектурные детали (управление памятью, машинные инструкции), позволяя программистам оперировать концептами, релевантными предметной области. При этом, в отличие от примера Л. Флориди, где выбор между УА покупателя и дегустатора вина ситуативен, в разработке ПО доминирует императив перехода к более высоким УА. Современные приложения для машин общего назначения требуют абстракций, которые не только маскируют аппаратную сложность, но и обеспечивают семантически насыщенное представление моделируемых сущностей.

Степень «высоты» УА языка программирования определяется двумя критериями: объемом скрываемых аппаратных деталей и способностью предоставлять интуитивные аналоги объектов предметной области. Этот дуализм отражает эволюцию языков – от низкоуровневых конструкций, привязанных к физической архитектуре, до высокоуровневых парадигм, где доминирует декларативное описание взаимодействий. Таким образом, если УА Л. Флориди – это инструмент эпистемологического упрощения, то в компьютерных науках абстракция становится технологическим императивом, связывающим философию проектирования с инженерной реализацией.

Специалисты в области компьютерных наук часто сравнивают языки программирования по их выразительности и применимости для различных задач. Если рассматривать язык как единый уровень абстракции, то анализ взаимосвязей между языками может быть организован через концепцию «градиента абстракций» (ГА), также предло-

женную Л. Флориди. Он вводит понятие «модерируемого градиента», который объединяет УА с «поведением» – предикатом, ограничивающим допустимые значения типизированных переменных [5, p. 310]. Например, вино не может одновременно быть белым и обладать высокой танинностью, что задает поведенческие ограничения для соответствующего УА.

Применяя эту концепцию к языкам программирования, их можно трактовать как модерируемые УА, где поведение определяется синтаксическими и семантическими правилами. Однако возникает противоречие: типизированные переменные (например, целочисленные или логические) имеют явные ограничения, тогда как функции, хотя и подчиняются синтаксическим правилам, демонстрируют поведение, несводимое к статическим ограничениям. Это объясняется различием целей: модель Л. Флориди ориентирована на информационное моделирование систем с четким описанием изменений состояния, тогда как разработчики языков программирования создают инструменты для реализации вычислительных объектов различной выразительности.

Для устранения этого противоречия необходимо расширить концепцию поведения, включив в нее как статические аспекты, так и динамические. В объектно ориентированных языках поведение метода определяется не только его сигнатурой, но и взаимодействием с состоянием объекта. Эволюция от ассемблера к языкам высокого уровня отражает градиент абстракций, где каждый уровень скрывает технические детали, усиливая выразительность.

Возвращаясь к концепции градиента абстракций, следует отметить, что данный конструкт формируется на основе множества УА, каждый из которых формализует масштаб или степень детализации модели. Согласно Л. Флориди, ГА предоставляет методологический инструмент для варьирования УА, позволяя проводить наблюдения на разных уровнях абстракции [5, p. 311]. Практическая значимость такого подхода иллюстрируется примером из области виноделия: поскольку процессы дегустации и коммерческой оценки вина опираются на различные УА, их объединение в рамках единого ГА обеспечивает корреляцию аспектов, наблюдаемых на каждом уровне. Таким образом, если УА характеризуется предикатами, определен-

ными для его наблюдаемых объектов, то ГА требует установления явных связей между каждой парой УА. Л. Флориди формализует эти связи с помощью декартовой системы координат, дополняя ее условиями согласованности, которые гарантируют непротиворечивость взаимодействия уровней. Однако данные условия обладают минимальной строгостью и не задают содержательных отношений между УА. Введение дополнительных ограничений позволяет выделить специализированные типы ГА, такие как «непересекающиеся ГА», где попарные УА не имеют общих наблюдаемых объектов, и «вложенный ГА», предполагающий линейное упорядочивание уровней, при котором значимые отношения возникают исключительно между соседними парами [5, p. 312].

Вложенные ГА полезны, потому что они могут «описывать сложную систему точно на каждом уровне абстракции и постепенно более точно» [5, p. 313], как в нейробиологических исследованиях, которые начинаются с изучения функций областей мозга в целом, а затем переходят к рассмотрению отдельных нейронов. Интересно отметить, что в то время как градиент во вложенном ГА идет от более абстрактного к более конкретному, градиент, применяемый в абстракции языка программирования, движется в ортогональном направлении, а именно от более ориентированного на машины к более ориентированному на мир.

Иными словами, вложенные ГА часто создаются для более точного анализа, в то время как новые уровни абстракции, предлагаемые языками программирования в области компьютерных наук, предоставляют более широкие возможности для синтеза при построении программных объектов еще большего объема, чем те, которые были доступны ранее. Содержание вычислительных объектов программиста расширяется по мере того, как все больше деталей, лежащих в основе машинного представления, скрыто от программиста выбранным языком реализации. Это особенно очевидно при рассмотрении возможностей абстрагирования данных языков более высокого уровня.

Языки более низкого уровня, такие как языки ассемблера, предлагают базовые типы данных, такие как числа, символы и строки. Эти типы сами по себе являются абстракциями для встроенных схем компьютера общего назначения. Если программист хочет написать,

например, веб-браузер на языке ассемблера, он должен тщательно реализовать высокоуровневое понятие, такое как коммуникационный сокет, на машинном языке – числа, символы, строки и т.д. Это скудный словарный запас, требующий кропотливого и трудоемкого программирования, которое не имеет отношения к основным проблемам просмотра веб-страниц. Сотни или, возможно, тысячи строк кода могут потребоваться только для выполнения того, что представляет собой учет данных на машинном уровне. Используя язык более высокого уровня, такой как *C*, программист может воспользоваться преимуществами более сложных типов, таких как структуры, которые могут автоматически группировать основные типы машин, и указатели, которые могут взять на себя вычисление адресов памяти, что значительно облегчает работу с этими основными типами машин. Но продолжая использовать языки более высокого уровня, такие как *C++* или *Java*, программисты могут расширить свой язык программирования, включив в него типы, значения которых сами по себе являются коммуникационными сокетами. Нет необходимости «подстраивать» высокоуровневое понятие коммуникационного сокета с использованием низкоуровневых типов программирования; коммуникационный сокет на самом деле является типом данных в этих языках более высокого уровня.

Вполне возможно, что для языков программирования и типов данных можно определить ГА, который соответствовал бы модели Л. Флориди, при условии, что отношения между УА, составляющими градиент, точно отражают природу сокрытия информации, от которой так сильно зависит разработка программного обеспечения. На самом деле концепция ГА может быть полезна для характеристики отношений между различными «парадигмами» языков программирования. Например, фундаментально императивный и машинно ориентированный характер программирования на языке ассемблера и чисто функциональный характер языков *Lisp* или *Scheme* могут помещать их в непересекающийся ГА, в то время как историческая эволюция языка *C* в *C++* предполагает для них вложенный ГА.

Какой бы ни была парадигма программирования, скрывая сложность работы с типами данных более низкого уровня, скрытие информации, обеспечиваемое языками высокого уровня, дает программис-

там и ученым выразительность и возможности для создания объектов более высокого уровня, которые заполняют все более сложные миры, созданные ими самими. Такие миры демонстрируют своего рода информационный «структурный реализм». Структурный реализм (СР) придает первостепенное значение отношениям, существующим между изучаемыми объектами, поскольку именно они, а не сами отношения облегчают объяснение, инструментирование и прогнозирование в рамках исследуемой системы [5, p. 220]. Что делает СР информативным, так это то, что вопросы об онтологическом статусе отношений могут быть отложены в сторону в пользу минимальной, информационной концепции объектов, несущих отношения. Чтобы объяснить эту идею, привлекается концепция «объекта» из объектно ориентированного программирования. Поскольку такие объекты инкапсулируют как состояние (множество наблюдаемых объектов в УА), так и поведение, они являются парадигмальными кандидатами на роль структурных объектов, охватываемых СР. Но будучи абстракциями, они избегают каких-либо онтологических требований к субстанции или материалу при описании структурных объектов.

Онтологически ни к чему не обязывающий характер информационно-структурного реализма (ИСР) Л. Флориди проявляется, когда он использует классификацию типов структурализма, предложенную Б. ван Фраассеном [16], и относит ИСР к категории «промежуточного» структурализма: ИСР не является ни радикальным, ни умеренным. Вместо этого «структура, описанная наукой, действительно имеет носителя, но этот носитель вообще не имеет никаких других признаков» [5, p. 221]. Однако информационные объекты обладают множеством функций, даже помимо важных связей, которые они имеют с другими объектами. Например, у директории на дисковом пространстве есть размер и местоположение, помимо связей с родительской папкой, в которой она находится, и подпапками, которые она содержит. Действительно, объекты ООП структурированы, но они не являются простыми отношениями; они богаты атрибутами – на языке ООП это обозначает нереляционные свойства. Программист мог бы, конечно, заменить атрибуты объекта другими объектами, чтобы объект состоял только из связей с другими объектами, но в конечном итоге эти другие объекты должны иметь атрибуты, кото-

рые не являются реляционными. Таким образом, не все объекты могут быть безликими, чего, по-видимому, желает Л. Флориди.

Современная эпистемология частично занимается тем, как мы используем наш чувственный опыт для получения непосредственных знаний об отдельных объектах, процессах и событиях в физическом мире посредством взаимодействия с ним нашего собственного тела. Но, как замечает С. Пинкер, «если бы наши знания на этом заканчивались, у нас не было бы средств эффективно взаимодействовать с миром. Знания, которые мы приобретаем благодаря наукам, начинаются только тогда, когда мы замечаем закономерности в ходе событий» [1, с. 15]. Когда утверждение о такой закономерности не допускает исключений, например что вода при давлении, установленном на уровне моря, закипает при температуре 100 градусов, это называется законом природы.

В аналитической философии XX в. существует богатая традиция попыток точно определить, что такое научная «закономерность». Однако в целом согласились, что как минимум утверждение является закономерным, если оно универсально (т.е. имеет всеобщий характер), не ограничено в пространстве или времени и неслучайно или необходимо [10; 11]. Когда знания обладают этими свойствами, они могут быть использованы как для объяснения, так и для прогнозирования конкретных наблюдаемых явлений и происшествий. Способность предсказать, что произойдет в той или иной ситуации, позволяет нам контролировать будущие события с сопутствующей такому контролю силой.

Информатика, не изучает природу, по крайней мере ту природу, которую изучают физика, химия или биология. Она изучает, конечно, информационные объекты, и большинство разработчиков программного обеспечения рассматривают свою реальность, в согласии с Л. Флориди, как независимую от разума и состоящую из структурных объектов, которые не являются ни вещественными, ни материальными, но информационными. Если информатика признается наукой, изучающей информационные процессы и системы, то какие эмпирические или фундаментальные законы лежат в ее основе? В чем заключается специфика этих законов по сравнению с естественными науками и как они проявляются в исследованиях ученых?

Некоторые утверждения, например знаменитый закон Г. Мура [14], являются обоснованными предсказаниями будущего технологий. Закон Мура – эмпирическое наблюдение об удвоении транзисторов на кристалле микросхемы каждые два года. Несмотря на многолетнюю точность, это не фундаментальный закон природы, а результат технологического прогресса, ограниченного физикой (атомарный масштаб, квантовые эффекты). Его «действие» прекратится, когда инженерные решения исчерпают себя, что подтолкнет ИТ-индустрию к поиску альтернатив – от трехмерных чипов до квантовых вычислений. Таким образом, закон Мура – скорее индустриальный ориентир, чем научная константа.

Другие предполагаемые законы, имеющие отношение к будущему технологий, уже доказали свою ложность. Закон Г. Гроша [7] гласил, что стоимость вычислений зависит только от квадратного корня из увеличения скорости, и поэтому Грош поддерживал разработку крупных суперкомпьютеров. Но произошло обратное: экономия за счет масштаба в большинстве случаев достигается посредством кластеризации большого количества обычных процессоров и дисковых накопителей.

Если законы Мура и Гроша основаны на технологиях, то другие утверждения, по-видимому, воплощают в себе если не законы природы, то законы вычислений. Например, Алан Тьюринг продемонстрировал, что невозможно написать процедуру, которая могла бы определить, будет ли та или иная процедура остановлена или будет выполняться бесконечно. Такое утверждение удовлетворяет приведенным выше критериям закономерности, а именно общности, пространственно-временной независимости и неслучайности, но это утверждение математической теоремы, а не эмпирического закона, подтверждаемого наблюдением.

Другие утверждения, которые стали известны как «законы», на самом деле также являются математическими соотношениями, применяемыми к задачам в области компьютерных наук. Закон Г. Амдаля [2], например, дает верхнюю границу ускорения, которое можно ожидать при попытке распараллелить последовательный алгоритм, запустив его одновременно на определенном количестве процессоров. Аналогичным образом закон Дж. Густафсона [8] модифицирует закон

Г. Амдаля, устраняя ограничение на фиксированное число процессоров. Каждый из этих законов выводится априори из абстрактных понятий, а не апостериорно из наблюдений.

Существует еще один важный набор математических законов, которыми в основном интересуются специалисты по информатике. Это ограничения пространства и времени, которые накладываются на вычислительные процессы из-за присущей им сложности. Точно так же, как законы Амдаля и Густафсона используют формальные математические аргументы для определения пределов ускорения, получаемого за счет использования нескольких процессоров, специалисты по информатике используют математические методы для классификации отдельных алгоритмов, например алгоритмов поиска в различных типах структур данных, с точки зрения объема памяти, требуемого для хранения структур данных, и количества используемых ресурсов. Важно отметить, что этот анализ основан не на реальном физическом исполнении алгоритмов с помощью программ, выполняемых на физических машинах, а на анализе алгоритмов как абстрактных математических объектов. При этом не проводится никаких эмпирических исследований. На самом деле типичная цель такого анализа – определить, может ли данное вычисление быть выполнено в разумных пределах пространства и времени, независимо от технологических прогнозов относительно скорости будущих машин. Может показаться, что такие утверждения об абстрактных вычислительных объектах удовлетворяют вышеуказанным критериям закономерности, но эти утверждения подкрепляются формальными доказательствами, а не эмпирическими исследованиями.

Так можно ли в каком-то смысле сказать, что информатика «открывает» научные законы в эмпирическом смысле? Можно утверждать, что специалисты по информатике не открывают законы, они должны их создавать. Когда программист определяет абстрактную процедуру сортировки абстрактного списка объектов, используя операции абстрактной машины, это происходит потому, что данная процедура будет реализована посредством сложной и примечательной цепочки электронных событий на данной машине в физическом мире. Эти события должны быть точно предсказаны, чтобы программа служила той цели, для которой она была предназначена. Чтобы это про-

изошло, такие электронные процессы должны подчиняться законам. Но описывать эти законы на физическом уровне электронов и микросхем было бы бессмысленно, поскольку физические законы электромагнетизма на молекулярном уровне не имеют отношения к задаче программиста, которая заключается в описании на формальном языке машины модели взаимодействия абстрактных объектов, таких как переменные, процедуры и структуры данных. Электронные процессы разворачиваются исключительно по заданным алгоритмам, их логика определяется не природными законами, а правилами, которые программист конструирует в абстрактной среде. Программисты наделяют абстрактные системы свойствами естественных законов – это придает вычислительным процессам объяснительную и прогностическую силу. Поэтому информатика во многом строится на метафорах: от «памяти» до «потоков данных».

В исследованиях [3; 15] анализируется, как метафоры позволяют ученым-компьютерщикам применять научные методы к абстрактным объектам. Например, хотя естественного «закона сохранения информации» не существует, сетевые программисты делают так, чтобы все работало, как если бы он существовал, разрабатывая алгоритмы обнаружения и исправления ошибок, гарантирующие, что биты не будут потеряны во время передачи. Закон сохранения информации в сетевых системах опирается на метафору «потока»: хотя биты данных не обладают физическим движением, как воздух или вода, их условное моделирование в виде потока позволяет специалистам формализовать математические условия для расчета пропускной способности сети и проектировать соответствующие алгоритмы и аппаратные решения. Эта метафора служит инструментом абстракции, преобразующим интуитивные представления в строгие инженерные параметры. Метафора потока – частное проявление более общей концепции движения, ключевой для информатики. Физически в компьютере движение сводится к работе вентиляторов и дисководов, но в абстрактных системах оно повсеместно: переходы состояний программ, обработка исключений, манипуляции с данными, управляющие циклы. Это закономерно: ядро информатики составляет моделирование динамических взаимодействий, где метафоры движения формализуют поведение нематериальных процессов.

Но «движение», происходящее в абстрактном мире информатики, было бы хаотичным, если бы его каким-то образом не ограничивали, и поэтому мы устанавливаем ограничения, знакомые нам по физическому миру. В случае сетевого программирования мы заимствуем из физики закон сохранения информации. Для программирования в целом мы заимствуем физические законы движения, чтобы представить законы вычислений в виде инвариантов.

Точно так же, как законы природы не допускают исключений, когда программисты устанавливают законы для своих абстрактных миров, они должны быть уверены, что эти законы не допускают изменений. Цель описательного закона природы – обнаружить и описать то, что есть. Задача предписывающего закона вычисления состоит в том, чтобы установить, что должно неизменно выполняться во время выполнения вычисления. Вот несколько примеров свободно сформулированных предписывающих законов: бит четности должен быть равен исключающему или одному из оставшихся битов, что используется в коде проверки ошибок; индекс массива не должен выходить за пределы диапазона [0...n–1], где n – размер массива, чтобы избежать ссылок за пределы массива; сегменты массива всегда должны находиться в определенных отношениях друг к другу – для суммирования или сортировки массива.

Эти предписания определяют ограничения, которые должны соблюдаться в соответствующих вычислительных средах. Когда ограничение сохраняется на протяжении всего вычислительного взаимодействия, например во время вызова процедуры или итерации цикла, оно становится инвариантом. Инварианты обладают двойной ролью. Во-первых, они служат дидактическим инструментом для объяснения алгоритмов [6], упрощая их понимание даже для начинающих. Во-вторых, они демонстрируют универсальность вычислительных законов: будучи определены для алгоритмов как абстрактных сущностей, инварианты формализуют принципы, которые управляют любой конкретной реализацией этих алгоритмов. Таким образом, инварианты связывают теоретическую абстракцию с ее практическим воплощением, выступая мостом между математической логикой и инженерной практикой.

Важно различать роль инварианта в абстрактном вычислительном процессе и роль аксиомы в формальной модели. Инварианты – это то, что мы хотим сделать истинным, в то время как аксиомы считаются истинными, если модель не показывает, что они противоречивы. Но несмотря на то, что мы стремимся к тому, чтобы инварианты были верными в соответствии с требованиями информатики, они часто нарушаются. Это объясняется двумя ключевыми особенностями. Во-первых, инварианты существуют на уровнях абстракции, которые невозможно напрямую реализовать в коде: программирование часто требует промежуточных шагов, временно нарушающих инварианты, например модификации данных перед их валидацией, и эти инварианты затем восстанавливаются. Во-вторых, инварианты могут намеренно нарушаться для адаптации алгоритмов или структур данных к новым задачам, например это оптимизация кеширования или расширение функциональности системы. Оба сценария иллюстрируют, как теоретические ограничения взаимодействуют с практической гибкостью в разработке программного обеспечения.

Сложные вычислительные процессы могут быть реализованы путем применения сложных инвариантов, которые сами по себе могут быть созданы путем адаптации более простых инвариантов. Мощь инвариантов также можно увидеть в знаменитом алгоритме быстрой сортировки, разработанном Ч. Хоаром [9]. Этот алгоритм основывается на рекурсивном разделении массива на сегменты, где опорный элемент занимает финальную позицию, а элементы меньше него располагаются слева, а больше или равные – справа. Этот процесс повторяется для каждого подсегмента, пока их размер не станет минимальным, обеспечивая сортировку. Ключевой механизм – поддержание инвариантов при разбиении: изначально все элементы, кроме опорного, помещаются в «необработанную» зону, которая постепенно сокращается по мере распределения элементов в зоны «меньше опорного» и «больше или равно». Когда необработанные элементы исчерпаны, опорный элемент фиксируется между подсегментами, после чего алгоритм рекурсивно применяется к левой и правой частям. Эффективность метода зависит от скорости разбиения и выбора опорного элемента, что минимизирует глубину рекурсии и общее время работы.

Любому, даже Ч. Хоару, было бы трудно представить себе этот алгоритм без помощи направляющего инварианта. Программисты начинают с инвариантов как с предписывающих закономерностей, а затем пытаются создать абстрактные миры, которые им подчиняются. Когда абстрактный процесс поддерживает тщательно заданный инвариант, его конкретная реализация будет вести себя так, как если бы она подчинялась описательному вычислительному закону. То есть его поведение будет предсказуемым, управляемым и выверенным для своей цели. Таким образом, инварианты в программировании – аналог законов природы: как планеты подчиняются гравитации, так программа, соблюдающая инварианты, действует предсказуемо. Они задают правила, ограничивая возможные состояния системы и превращая хаос в контролируемый процесс, где входы однозначно определяют результат.

Фрэнсис Бэкон писал: «Природу побеждают, только повинуясь ей». Это применимо и к информатике, но при этом разработчики программ устанавливают законы для программ, а затем должны обеспечить, чтобы программы подчинялись этим законам. Затем законы становятся ограничениями для программиста. Но это не ограничения в политическом смысле. Опасность политического беззакония носит внешний характер: ваша неограниченная свобода представляет угрозу для моей свободы, и наоборот. Но опасность беззакония в области компьютерных наук носит внутренний характер: наш разум нуждается в ограничениях, чтобы анализировать последствия программных решений. Например, для программиста крайне важно восстановить инвариант для каждой итерации цикла.

Хотя ограничения в целом ограничивают свободу, в случае программирования они значительно облегчают достижение цели, разбивая решение задачи на последовательность небольших шагов, каждый из которых определяется инвариантом. Метод временного нарушения инвариантов с последующим восстановлением, применяемый программистами для работы с массивами, оказывается продуктивным и в других областях. Например, рассмотрим структуру данных – бинарное дерево поиска (БДП). БДП обеспечивает эффективный поиск данных по ключу аналогично телефонному справочнику: ключи – имена людей связаны с соответствующими значениями – адресами,

номерами телефонов. Инвариант БДП требует, чтобы для любого узла левое поддерево содержало ключи меньше его значения, а правое – больше. Нарушение этого инварианта (например, при вставке элемента) всегда временное и устраняется корректировкой структуры дерева, что гарантирует сохранение его функциональности. Этот принцип демонстрирует универсальность подхода: управление инвариантами позволяет создавать гибкие, но предсказуемые системы.

Вычислительная реальность, по своей сути информационная, существует в балансе между закономерностями и свободой их трансформации. Инварианты здесь выступают как самоустанавливаемые «законы»: они задают структурные рамки, обеспечивая предсказуемость систем, но их нарушение – не хаос, а творческий акт, открывающий путь к эволюции алгоритмов. Например, временный отказ от инварианта в бинарном дереве поиска позволяет реорганизовать его структуру, сохраняя функциональность на более высоком уровне абстракции.

Эта динамика отражает суть компьютерных наук: уровни абстракции служат инструментом управления сложностью, где каждый слой инкапсулирует свои инварианты, подобно объектам в объектно ориентированном программировании. Л. Флориди, исследуя информационную онтологию, справедливо отмечает, что подобные методологии – не просто технические приемы, а форма эпистемологического прогресса. Даже если ООП уступит место новой парадигме, движение к более высоким абстракциям останется трендом, ибо оно отражает фундаментальную потребность – дробить реальность на управляемые информационные единицы, сохраняя их взаимосвязи.

Вопрос в том, станет ли этот процесс метафорой для понимания всей реальности. Если физический мир – тоже система уровней абстракции (квантовый, классический, космологический), то инварианты природы, подобно программным, могут оказаться не закономерностями, а условными правилами, допускающими перетолкование в иных контекстах. Но пока ясно одно: в информатике свобода переопределять инварианты, не разрушая систему, – не хаос, а доказательство того, что порядок и творчество могут сосуществовать, порождая новые формы закономерностей и предсказуемости.

Литература

1. *Пинкер С.* Просвещение продолжается: В защиту разума, науки, гуманизма и прогресса. М.: Альпина нон-фикшн, 2021.

2. *Amdahl G.M.* Validity of the single processor approach to achieving large scale computing capabilities // Proceedings of the Spring Joint Computer Conference. Reston, Va.: AFIPS Press, 1967. P. 483–485.

3. *Erickson T.D.* Working with interface metaphors // Readings in Human-Computer Interaction. Morgan Kaufmann, 1995. P. 147–151.

4. *Floridi L.* A defence of informational structural realism // Synthese. 2008. Vol. 161. P. 219–253.

5. *Floridi L.* The method of levels of abstraction // Minds and Machines. 2008. Vol. 18. P. 303–329.

6. *Gries D.* The Science of Programming. N.Y.: Springer, 1981.

7. *Grosch H.R.* High speed arithmetic: The digital computer as a research tool // Journal of the Optical Society of America. 1953. Vol. 43, iss. 4. P. 306–310.

8. *Gustafson J.L.* Reevaluating Amdahl's law // Communications of the ACM. 1988. Vol. 31, No. 5. P. 532–533.

9. *Hoare C.A.R.* Quicksort // Computer Journal. 1962. Vol. 5. P. 10–15.

10. *Knox E.* Physical relativity from a functionalist perspective // Studies in History and Philosophy of Science. Part B: Studies in History and Philosophy of Modern Physics. 2019. Vol. 67. P. 118–124.

11. *Lange M.* Because Without Cause: Non-Causal Explanations in Science and Mathematics. Oxford: Oxford University Press, 2016.

12. *MacLane S.* Mathematics Form and Function. Springer Science & Business Media, 2012.

13. *Milner R.* Communication and Concurrency. Englewood Cliffs: Prentice Hall, 1989. Vol. 84.

14. *Moore G.E.* Cramming more components onto integrated circuits // Electronics. 1965. April 19. P. 114–117.

15. *The Philosophy* of Information Quality / Ed. by L. Floridi, P. Illari. Cham: Springer, 2014. (SYLI, vol. 358).

16. *Van Fraassen B.C.* Structure: Its shadow and substance // The British Journal for the Philosophy of Science. 2006. Vol. 57, No. 2. P. 275–307.

References

1. *Pinker, S.* (2021). Enlightenment Now: The Case for Reason, Science, Humanism, and Progress. Moscow, Alpina Non-fiction Publ. (In Russ.).

2. *Amdahl, G.M.* (1967).Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference. Reston, Va., AFIPS Press, 483–485.

3. *Erickson, T.D*. (1995). Working with interface metaphors. In: Readings in Human-Computer Interaction. Morgan Kaufmann, 147–151.

4. *Floridi, L*. (2008). A defence of informational structural realism. Synthese, 161, 219–253.

5. *Floridi, L*. (2008). The method of levels of abstraction. Minds and Machines, 18, 303–329.

6. *Gries, D*. (1981). The Science of Programming. New York, Springer.

7. *Grosch, H.R*. (1953). High speed arithmetic: The digital computer as a research tool. Journal of the Optical Society of America, Vol. 43, No. 4, 306–310.

8. *Gustafson, J.L*. (1988). Reevaluating Amdahl's law. Communications of the ACM, Vol. 31, No. 5, 532–533.

9. *Hoare, C.A.R*. (1962). Quicksort. Computer Journal, 5, 10–15.

10. *Knox, E*. (2019). Physical relativity from a functionalist perspective. Studies in History and Philosophy of Science, Part B: Studies in History and Philosophy of Modern Physics. 2019, 67, 118–124.

11. *Lange, M*. (2016). Because Without Cause: Non-Causal Explanations in Science and Mathematics. Oxford, Oxford University Press.

12. *MacLane, S*. (2012). Mathematics Form and Function. Springer Science & Business Media.

13. *Milner, R*. (1989). Communication and Concurrency, Vol. 84. Englewood Cliffs, Prentice Hall.

14. *Moore, G.E*. (1965). Cramming more components onto integrated circuits. Electronics, April 19, 114–117.

15. *Floridi, L. & P. Illari* (Eds.). (2014). The Philosophy of Information Quality. SYLI, Vol. 358. Cham, Springer.

16. *Van Fraassen, B.C*. (2006). Structure: its shadow and substance. The British Journal for the Philosophy of Science, Vol. 57, No. 2, 275–307.

Информация об авторе

Разумовский Алексей Игоревич – Институт проблем управления им. В.А. Трапезникова РАН (117997, Москва, ул. Профсоюзная, 65).

razumowsky@yandex.ru

Information about the author

Razumowsky, Alexey Igorevich – Trapeznikov Institute of Control Sciences, Russian Academy of Sciences (65, Profsoyuznaya St., Moscow, 117997, Russia).

razumowsky@yandex.ru

Дата поступления 24.09.2025.
Принята к публикации 02.02.2026.