DOI: 10.15514/ISPRAS-2026-38(4)-4

Системы эффектов как механизм управления контекстными свободными переменными: единый взгляд

А. С. Стоян, ORCID: 0009-0004-6935-1447 <a.stoyan@hse.ru>

Национальный исследовательский университет «Высшая школа экономики», Россия, 194100, г. Санкт-Петербург, ул. Кантемировская, д. 3А.

Аннотация. Системы эффектов поднимают сведения о вычислительных эффектах программы на уровень типов: они уточняют границы абстракций и позволяют компилятору проверять, что эффектное вычисление выполняется в поддерживающем его контексте. Тем не менее статическое отслеживание эффектов пока не стало повсеместной практикой: ведущие подходы – строки эффектов, возможности (capabilities) и модальные типы – обычно описываются с помощью различного технического аппарата, и проектировщику языка трудно увидеть общую картину пространства решений. Мы предлагаем объединяющий взгляд: система эффектов – это механизм управления контекстными свободными переменными вычисления – зависимостями, которые не передаются обычными аргументами, а должны разрешаться контекстом исполнения. Каждая такая переменная проходит *жизненный цикл* из двух фаз: она возникает ожидающей – с неразрешённым требованием к контексту; после разрешения зависимость может сохраниться внутри значения – либо буквально, посредством захвата свидетельства, либо в широком смысле, как зависимость от контекста построения. Семейства систем эффектов различаются тем, какие фазы и переходы этого цикла они поддерживают и как отражают их в типах. Классические строковые системы держат переменные ожидающими и не выражают захват. Системы на основе возможностей допускают захват, но удерживают захваченную переменную в предоставившем её контексте средствами escape-анализа. Модальные системы вместо запрета *повторно освобождают* утёкшую переменную, снова выставляя её в типе как требование к контексту. Сравнение семейств вдоль единой оси жизненного цикла показывает, что выбор поддерживаемых переходов предопределяет остальные черты дисциплины – от необходимого вида полиморфизма до мест сосредоточения аннотаций – и тем самым даёт проектировщику языка карту пространства решений.

Ключевые слова: системы эффектов; вычислительные эффекты; контекстные свободные переменные; свободные переменные; неявные параметры; escape-анализ; возможности; строки эффектов; модальные типы; системы типов; контекстный полиморфизм.

Для цитирования: Стоян А.С. Системы эффектов как механизм управления контекстными свободными переменными: единый взгляд. Труды ИСП РАН, 2026, том 38, вып. 4, часть 1, с. 69–88. DOI: 10.15514/ISPRAS-2026-38(4)-4.

Effect Systems as a Mechanism for Managing Contextual Free Variables: A Unifying Perspective

A. S. Stoyan, ORCID: 0009-0004-6935-1447 <a.stoyan@hse.ru>

National Research University Higher School of Economics, 3A, Kantemirovskaya Str., Saint Petersburg, 194100, Russia.

Abstract. Effect systems lift information about a program's computational effects to the type level: they sharpen abstraction boundaries and let the compiler verify that an effectful computation runs in a context that supports it. Nevertheless, static effect tracking has not yet become common practice: the leading approaches – effect rows, capabilities, and modal types – are usually described using different technical machinery, and a language designer struggles to see the overall design space. We propose a unifying perspective: an effect system is a mechanism for managing the contextual free variables of a computation – dependencies that are not passed as ordinary arguments but must be resolved by the execution context. Such a variable goes through a *lifecycle* of two phases: it arises pending, with an unresolved contextual requirement; once resolved, the dependency can be retained inside a value – either literally, by capturing a witness, or more broadly, as a dependency on its construction context. Families of effect systems differ in which phases and transitions of this lifecycle they support and how they reflect them in types. Classical row-based systems keep the variables pending and cannot express capture. Capability-based systems allow capture but confine a captured variable to the context that provided it by means of escape analysis. Modal systems, instead of forbidding the escape, *uncapture* an escaping variable, re-exposing it in the type as a contextual requirement. Comparing the families along this single lifecycle axis shows that the choice of supported transitions predetermines the remaining traits of the discipline – from the kind of polymorphism required to the placement of annotations – thus giving the language designer a map of the design space.

Keywords: effect systems; computational effects; contextual free variables; free variables; implicit parameters; escape analysis; capabilities; effect rows; modal types; type systems; contextual polymorphism.

For citation: Stoyan A.S. Effect Systems as a Mechanism for Managing Contextual Free Variables: A Unifying Perspective. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 4, part 1, pp. 69-88 (in Russian). DOI: 10.15514/ISPRAS-2026-38(4)-4.

1. Введение

Программирование в значительной степени сводится к управлению сложностью: разработчик стремится сосредоточиться на отдельном фрагменте поведения, не держа в голове всю программу целиком. Абстракция и разделение ответственности – ключевые приёмы: различные обязанности делегируются разным сущностям, таким как функции или модули. Часто в качестве такой сущности удобно выделять *контекст исполнения* вычисления, а *эффекты* возникают естественным образом как взаимодействия вычисления с этим контекстом [1].

Начнём с противоположного понятия – чистоты. Единственный наблюдаемый результат чистого вычисления есть его значение. Функцию называют чистой, если она возвращает одинаковый результат на одинаковых аргументах, а её применение является чистым вычислением. Программирование с чистыми функциями обычно считается хорошей практикой: композиция чистых функций чиста, семантика кода прозрачна, а системы типов работают хорошо, обеспечивая частичную спецификацию, безопасность и ясность абстракций.

Однако писать и сопровождать нетривиальную программу, пользуясь лишь чистыми вычислениями, трудно. Всё состояние и поток управления должны контролироваться явно с помощью аргументов и результатов функций. Приведём пример ручного распространения ошибок – функция возвращает `Result`, и каждый вызов сопровождается проверкой:

```
fun readInts(n: Int): Result<List<Int>, Error> {
    if (n == 0) { return Ok(Nil()) }
    let x = readInt()
    if (x is Err) { return x }
    let xs = readInts(n - 1)
    if (xs is Err) { return xs }
    return Ok(Node(x.value, xs.value))
}
```

Нужен инструмент управления сложностью – нечто, что скрывает подобные несущественные детали. Вычислительные эффекты и есть такой инструмент. Эффектом мы будем называть взаимодействие вычисления с контекстом исполнения; вычисление обращается к контексту напрямую, без церемоний, которыми полон чистый код. Контекстом может служить система времени выполнения, предоставляющая сервисы ОС, изменяемые ячейки памяти, механизмы передачи управления, ввод-вывод и тому подобное. *Эффектная операция* (effectful operation) – это конструкция, порождающая эффект. Например, механизм исключений позволяет вычислению сообщить об ошибке контексту – ближайшему обработчику. Рутинное распространение ошибок исчезает из кода – остаётся существо дела:

```
fun readInts(n: Int): List<Int> {
    if (n == 0) { return Nil() }
    return Node(readInt(), readInts(n - 1))
}
```

Некоторые подходы позволяют программисту определять собственные эффекты и контексты исполнения как библиотечный код. В частности, *обработчики эффектов* (effect handlers) – многообещающий подход, зарекомендовавший себя и за пределами сообщества чисто функциональных языков [2-4]. Обработчик задаёт область, в которой допустимы определённые эффектные операции, и интерпретирует их в терминах чистых значений и эффектов объемлющей области. Собственные эффекты служат для построения встроенных предметно-ориентированных языков [5], а такие подходы соревнуются в выразительности, производительности и удобстве композиции независимо определённых эффектов [1, 6-8].

Многие эффектные операции допустимы лишь в ограниченной области. Например, исключение можно бросить только в контексте соответствующего обработчика, а изменяемую ячейку, размещённую на стеке, нельзя использовать после завершения функции. Статический контроль над применением таких операций предотвращает множество ошибок времени выполнения и повышает надёжность программ. *Системы эффектов* как раз и являются механизмом обеспечения такого контроля на уровне типов. Кроме того, поскольку эффекты по своей природе неявны на уровне термов, об их наличии сложно судить по исходному коду: так, сигнатура эффектной версии `readInts` из показанного примера умалчивает о возможной ошибке. Поэтому всё же требуется некоторая степень явности – вторая задача систем эффектов. Это помогает задавать чёткие функциональные абстракции и не давать деталям реализации незаметно протекать в виде взаимодействий с контекстом.

Идея отслеживать эффекты в типах восходит к работе Лукассена и Гиффорда [9] и развившей её дисциплине типов и эффектов Талпина и Жувело [10]; показательно и обратное движение современного мейнстрима: OCaml 5 включил обработчики эффектов, вовсе не отслеживая их в типах [11], что делает вопрос о практичной системе эффектов лишь острее. Было предложено множество перспективных конструкций систем эффектов: полиморфные по строкам типы [12], возможности (capabilities) [13-14], модальные типы [15] и др. Однако каждый подход обладает своими достоинствами и недостатками, а общая картина пространства решений изучена недостаточно. При этом проектировщику языка важно иметь целостное представление о доступных вариантах, чтобы выбрать наиболее подходящее решение с учётом особенностей и ограничений целевого языка.

В настоящей работе мы предлагаем единый взгляд на эти, казалось бы, разрозненные подходы: *система эффектов – это механизм управления контекстными свободными*

переменными вычисления на уровне типов. Такие переменные не задаются обычными аргументами терма, но должны быть разрешены окружающим контекстом: обработчиком, ресурсом или возможностью. Этот взгляд, опирающийся на наблюдения из языка Scala [14, 16], позволяет рассматривать классическую систему типов и систему эффектов как две стороны одной задачи: первая имеет дело с переменными, которые разрешает *лексическое окружение*, вторая – с теми, которые разрешает *контекст исполнения*. Эти переменные, в свою очередь, проходят единый *жизненный цикл*, и каждое из рассматриваемых семейств систем эффектов оказывается выбором того, какие фазы и переходы этого цикла доступны и как они отражаются на уровне типов.

Настоящая работа – именно объединяющая перспектива, а не исчерпывающий обзор: охват сознательно сужен до трёх ведущих семейств. Наиболее буквально эта перспектива применима к эффектам, возникающим из взаимодействия с контекстом исполнения, – таким как исключения, ввод-вывод или изменяемое состояние. Термин, впрочем, употребляется и шире: эффектами называют и предикаты на вычислениях – например, расходимость или оценки стоимости; для них чтение через контекстные свободные переменные – скорее аналогия, чем буквальная модель.

Работа основана на нашем более раннем обзоре [17], но отличается от него по существу. Новыми являются сама модель жизненного цикла контекстной свободной переменной (рис. 1), понятие *повторного освобождения* (uncapturing) захваченной переменной и прочтение каждого семейства в терминах цикла (разделы 4-6). Новы также сравнительный анализ и синтез (разделы 7 и 8). Вклад работы можно резюмировать так:

- мы формулируем единую точку зрения на системы эффектов как на средство управления контекстными свободными переменными и описываем их *жизненный цикл*, вдоль фаз и переходов которого располагаются все рассматриваемые конструкции (раздел 3);
- мы заново рассматриваем три семейства современных систем эффектов – строковые (раздел 4), на основе возможностей (раздел 5) и модальные (раздел 6) – и показываем, какие переходы цикла поддерживает каждое из них;
- мы проводим сравнительный анализ семейств вдоль единой оси жизненного цикла и показываем, что выбор поддерживаемых переходов предопределяет и необходимый вид полиморфизма, и места сосредоточения аннотаций, а статическую дисциплину системы эффектов, понимаемой как управление контекстными свободными переменными, можно рассматривать как сочетание двух знакомых практикам механизмов – неявных параметров и escape-анализа на уровне типов (разделы 7 и 8).

2. Свободные переменные как воплощение зависимостей от контекста

В этом разделе мы показываем, что прагматические мотивы систем эффектов естественно возникают в практике программирования, и готовим почву для центральной идеи работы (раздел 3). Вместо того чтобы отталкиваться от устройства существующих систем, мы выведем требования к системам эффектов конструктивно – последовательным рефакторингом повседневного сценария. Для неформальных пояснений мы будем пользоваться примерами на вымышленном языке в духе Scala и Kotlin, вводя его конструкции по мере необходимости.

Рассмотрим сквозной пример: функция бизнес-логики выполняет запрос к базе данных. В первой версии она сама устанавливает соединение:

```
fun business1() {
    let db = Db.open(...)
    db.query("...")
}
```

У этого решения есть недостатки. Во-первых, зависимость от базы данных зашита в функцию, а не предоставляется контекстом: вызывающий код не может её варьировать – например, подменить в тестах. Во-вторых, функция производит наблюдаемые эффекты – обращения к базе данных, – которые являются частью её внешнего контракта, но никак не отражены в сигнатуре.

Вторая версия заметно лучше. Функция параметризована соединением, и место вызова может передать его конкретную реализацию:

```
fun business2(db: Db) {
    db.query("...")
}
fun caller() {
    let db = Db.open(...)
    business2(db)
}
```

Теперь поведение функции начинает определяться контекстом исполнения, а наблюдаемые эффекты явно выражены через параметр `db`. Однако за это приходится платить синтаксическим шумом в месте вызова: стоимость явной передачи растёт с числом зависимостей и глубиной стека – параметр протягивается через все промежуточные функции, включая те, что его не используют. Именно эта цена и склоняет разработчиков искать менее громоздкие решения.

Чтобы устранить синтаксический шум, в третьей версии обычный параметр заменяется *динамической свободной переменной* `?db`, получающей своё значение поиском в динамической области видимости вызова:

```
fun business3() {
    ?db.query("...")
}
fun caller() {
    let ?db = Db.open(...)
    business3()
}
```

Тем не менее отсутствие синтаксического шума достигается ценой утраты других преимуществ: теряется статическая проверка наличия и типа такой переменной, как и явное представление соответствующего эффекта в сигнатуре. Эти потери восполняют *системы эффектов*: они предлагают способы сохранить статическую безопасность, не вводя лишних церемоний на уровне термов. Конкретные решения мы рассмотрим в дальнейшем.

Зафиксируем главное требование к системе эффектов – *безопасность эффектов*: всякий эффект может выполняться лишь в контексте, который его поддерживает, иначе программа должна быть отвергнута. В терминах свободных переменных это означает, что каждая такая переменная должна быть разрешена объемлющим контекстом вызова.

Полезен и контроль за тем, что происходит с такой переменной после получения значения. Здесь различимы два события: сначала динамическая переменная `?db` разрешается в конкретное значение, а затем замыкание захватывает уже это значение – как *лексически связанную* свободную переменную – и может пережить контекст, который его предоставил:

```
fun leak(): () -> Unit {
    let db = ?db // разрешение: получаем значение
    let f =
        () => db.query("...") // лексический захват значения db
    return f // захватившее замыкание покидает leak
}
```


Сама по себе `leak` ещё безопасна – всё решает вызывающий код: если область, связавшая `?db`, завершится раньше вызова `f` и, скажем, закроет соединение, отложенный вызов обратится к уже закрытому соединению.

Поскольку многие эффекты допустимы лишь в ограниченной области, подобные захваченные переменные не должны утекать за её пределы – иначе безопасность эффектов будет потеряна, – и для контроля за этим системе эффектов, допускающей такой захват, нужна та или иная форма escape-анализа [18-19]. Взамен слежение за лексическим захватом позволяет не пробрасывать сведения о свободных переменных функции-аргумента в тип функции высшего порядка – эта техника называется *контекстным полиморфизмом* (contextual polymorphism) и будет рассмотрена далее (раздел 5).

3. Жизненный цикл контекстной свободной переменной

Предыдущее обсуждение по существу велось об особого рода свободных переменных. Назовём их *контекстными свободными переменными* и будем понимать их широко: это зависимости вычисления от окружающего контекста исполнения – обработчика эффекта, значения-возможности или ресурса, без которых вычисление не может быть безопасно запущено. Слово «переменная» здесь фиксирует не обязательное синтаксическое представление, а роль в программе: как свободная переменная терма должна быть разрешена окружением, так и эффектное вычисление требует поддержки от контекста. Обязанность контекста удовлетворить такую зависимость будем называть *контекстным требованием*.

Центральное наблюдение работы таково: контекстная свободная переменная может проходить *жизненный цикл* из двух фаз – *ожидания* и *захвата* (рис. 1). Переменная возникает *ожидающей*: тело функции нуждается в чём-то от контекста, но ещё не получило этого; системы эффектов отражают такое неразрешённое требование в типах. Затем требование *разрешается* окружающим контекстом – так в разделе 2 разрешалась переменная `?db`. Отложенное вычисление может сохранить разрешённую зависимость: буквально – *захватив* материализованное значение-*свидетельство* (witness) разрешения, или в широком смысле – став зависимым от контекста своего построения.

На перенос захваченной переменной за границу разрешившего её контекста дисциплина может ответить двумя способами: *удержанием* – запретив переменной покидать область, – либо *повторным освобождением* – переведя её обратно в фазу ожидания и снова отразив в типе как контекстное требование[1].

Заметим, что «разрешено» – не отдельная фаза, а лишь промежуточное, локально разрешённое состояние на переходе от ожидания к захвату: контекст уже разрешил требование, но зависимость ещё не стала частью значения, способного пережить текущую активацию, – такое состояние не требует отражения в типах.

Двум фазам этого цикла отвечают две классические дисциплины *связывания* свободных переменных. Ожидающая переменная в характерном случае связывается *динамически*: её значение (например, обработчик) определяется вызывающим контекстом, – хотя возможно и связывание по месту определения (раздел 4). Захваченная – *лексически*: зависимость зафиксирована по месту захвата – буквально замыканием или, в широком смысле, самим контекстом построения.

Отметим связь с известным понятием: неразрешённое контекстное требование – по существу *коэффект* в смысле работ о контекстной зависимости вычислений [20], каноническими примерами которой служат неявные параметры и динамическое связывание. Ортогональным вкладом настоящей работы является слой *захвата* с escape-анализом и повторным

[1] «Освобождение» – возвращение переменной статуса *свободной*, ожидающей; к освобождению памяти термин отношения не имеет.

освобождением, который теория коэффектов не выделяет как отдельное измерение (подробнее – раздел 9).

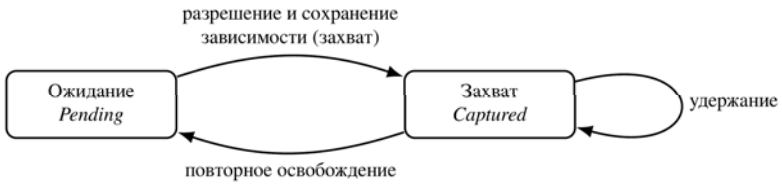

Рис. 1. Жизненный цикл контекстной свободной переменной.
Fig. 1. Lifecycle of a contextual free variable.

Оговоримся ещё раз: мы предлагаем объединяющую точку зрения, а не формальную классификацию и не доказательство выразительной эквивалентности всех систем. Подчеркнём: семейства мы будем различать не по тому, что происходит с переменной во время выполнения, а по тому, какие *переходы* цикла дисциплина поддерживает и как отражает их в типах.

В дальнейших разделах мы пользуемся этой оптикой, чтобы заново взглянуть на выбранные семейства систем эффектов; сравнительный итог – как выбор поддерживаемых переходов предопределяет облик дисциплины – сведён в разделе 7, а конструктивный – сборка статической дисциплины из механизмов двух фаз цикла – в разделе 8.

4. Строковые системы эффектов

Большинство систем эффектов представляет набор эффектов типами-строками[2] (row-типы) [21]. В этом разделе рассматриваются классические строковые системы, отличающиеся тем, что хранят в строках сведения обо *всех* внешне наблюдаемых эффектах функции. Метки – это имена эффектов (возможно, с типовыми аргументами), а строка – в первом приближении неупорядоченный набор меток (точная её алгебра варьируется от системы к системе). Записывать строку будем после типа результата, за косой чертой.

Вернёмся к сквозному примеру из раздела 2: доступ к базе данных становится эффектом `db`, и его метка в строке объявляет требование тела функции:

```
fun business(): Unit / {db} {
    query("...")
}
```

Операция `query`, как и переменная `?db` прежде, будет связана с конкретной реализацией из динамического контекста вызова.

Функция высшего порядка, вызывающая свой аргумент, но сама не удовлетворяющая его контекстных требований, производит по меньшей мере те же эффекты, что и он. Строковые системы выражают это *полиморфизмом по строкам* [21]: переменные типа могут обозначать строки и встраиваться в другие строки. Так, `map` повторяет строку эффектов своего аргумента через переменную `e`:

```
fun map<e, a, b>(f: (a) -> b / {e}, xs: List<a>): List<b> / {e}
```

В общем случае функция высшего порядка, вызывающая аргумент с заранее неизвестными эффектами и делегирующая их вызывающему контексту, должна быть параметризована переменной строки эффектов. Это существенно затрудняет плавную миграцию к такой системе эффектов и вносит значительный синтаксический шум; вывод эффектов (например,

[2] «Строка» здесь – перевод англ. row (строка-запись, как строка таблицы); со строками символов (string) термин не связан.

в языках программирования Koka и Flix) облегчает это бремя, но не устраняет полиморфизма по строкам из публичных сигнатур.

Пример строковой системы – проверяемые исключения Java [22]: язык статически удостоверяется, что все проверяемые исключения пойманы или явно объявлены в сигнатуре метода. Эта черта снискала печальную известность у программистов [23]: строка исключений – специальная (ad hoc) конструкция, допускающая параметризацию лишь фиксированным числом переменных-исключений (generic throws), но не переменной по произвольному их набору, поэтому метод высшего порядка, пробрасывающий заранее неизвестный набор проверяемых исключений, невыразим. Полноценные же строки лежат в основе систем эффектов Koka [12], Links [24], Flix [25] и Unison [26].

Чтобы удовлетворить требование `db`, воспользуемся, например, обработчиком эффектов. Так, функция `withDb` открывает соединение, интерпретирует с его помощью операцию `query` переданного вычисления и по завершении закрывает соединение:

```
fun withDb<e>(f: () -> Unit / {db | e}): Unit / {e} {
    let conn = Db.open(...)
    handle (f()) { op query(s) { resume(conn.exec(s)) } }
    conn.close()
}

withDb(() => business())
```

Вычисление `f` выступает клиентом: обращение к `query` передаёт управление обработчику, который выполняет запрос на открытом соединении и возвращает клиенту результат вызовом `resume`[3]. В сигнатуре `withDb` видно, что у `f` есть набор контекстных требований, включающий `db` (вертикальная черта расширяет строку переменной `e` этой меткой), `withDb` же удовлетворяет `db`, а остальные контекстные требования `e` делегирует вызывающему контексту[4].

С позиций предлагаемого взгляда строковая система делает явной фазу *ожидания*. Строка эффектов в сигнатуре есть типовой след ожидающих контекстных переменных тела функции, которые ещё не связаны: каждая метка – такая как `db` – обозначает эффект, поддерживать который обязан контекст исполнения. Сами метки при этом – не переменные терма, а имена соответствующих требований к контексту. В терминах жизненного цикла обращение к `query` в теле `f` – вхождение контекстной свободной переменной: обработчик динамически связывает его в своей области.

Не менее важно, чего эта дисциплина *не* позволяет: связать ожидающую переменную значением и затем захватить его – такой захват невыразим в самой типовой дисциплине. Связь операции с обработчиком устанавливается заново при каждом обращении, поэтому тип отложенного вычисления продолжает перечислять переменные как ожидающие, в какой бы контекст оно ни попало. Отсюда два следствия. Во-первых, строковым системам не нужен escape-анализ: пока свидетельства разрешения не материализуются в отслеживаемых значениях, утечка невозможна[5]. Во-вторых, именно поэтому и растёт объём аннотаций: ожидающая переменная не может «спрятаться» внутри значения, и её приходится повторять в типе каждой промежуточной функции – отсюда повсеместный полиморфизм по строкам и недоступность контекстного полиморфизма.

[3] Эффекты открытия и закрытия соединения считаем инкапсулированными внутри `withDb`: в сигнатуру входят лишь требования к внешнему контексту.

[4] Заметим, что область обработчика здесь намеренно совпадает с временем жизни соединения: одна лексическая область и разрешает требование `db`, и владеет ресурсом. Это совпадение станет существенным при обсуждении утечек (раздел 5).

[5] Имена обработчиков первого класса, материализующие свидетельство, возвращают потребность в контроле области (раздел 9).

Сближение строк с динамической дисциплиной связывания имеет и известные границы. Во-первых, существуют строковые системы с *лексически* связываемыми обработчиками: связь операции с обработчиком фиксируется по месту определения, а не поиском в динамическом контексте вызова [27]; туннелирование обработчиков аналогично сочетает строковую запись с лексической дисциплиной, решая проблему случайной обработки [28-29]. Во-вторых, техника передачи свидетельств (evidence passing) компилирует строковые эффекты Koka в явную передачу значений-свидетельств обработчиков [30]. Оба наблюдения не опровергают прочтение, а уточняют его. Строка говорит лишь, что требование ещё *не разрешено*, но не фиксирует, *как* оно разрешится: поиском в динамическом контексте вызова или по месту определения, лексически. Не закреплено ею и то, что запись требования должна оставаться меткой в строке: передача свидетельств систематически превращает её в явно передаваемое значение – свидетельство обработчика. Эта материализация, впрочем, происходит в целевом языке компиляции: программисту исходного языка свидетельство как значение недоступно, поэтому захват остаётся невыразимым в исходной дисциплине. Значит, семейства расходятся не в том, *что* они отслеживают – это всё та же контекстная переменная, – а в том, *как* они её записывают. Но у этого выбора есть цена, и раздел 7 её покажет: escape-анализ, вид полиморфизма, места сосредоточения аннотаций.

5. Системы на основе возможностей (capabilities)

Ранее эффекты возникали из вызовов «особых» функций – операций, – а их метки перечислялись в строковых типах. Теперь взглянем на отслеживание эффектов иначе: пусть эффекты выполняются только через «особые» объекты – *возможности* (capabilities), значения-свидетельства того, что контекст поддерживает соответствующий эффект; тогда система может отслеживать эти объекты вместо самих операций.

Вернёмся к сквозному примеру. Явная передача соединения аргументом (`business2`) статически безопасна, но шумна; динамическая переменная `?db` удобна, но небезопасна. Чтобы соединить достоинства обоих вариантов, аппроксимируем динамически связываемую переменную *неявным параметром* функции [31]: объявим группу таких параметров ключевым словом `context`; объявление `context let` в месте вызова делает значение доступным компилятору для автоматической подстановки:

```
context(db: Db)
fun business() { db.query("...") }

fun caller() {
    context let db = Db.open(...)
    business()
}
```

Неявные параметры типизируют контекстные свободные переменные функции: чтобы вызвать такую функцию, контекст обязан предоставить конкретные значения – те самые возможности[6].

Исключения и другие эффектные операции отслеживаются аналогично [16]: конструкцию `throw SomeException()` можно понимать как вызов функции, требующей неявного параметра типа `CanThrow<SomeException>`, а блок `try-catch` предоставляет свидетельства соответствующих типов. Встроенная поддержка для этого не обязательна: дисциплина возможностей – прежде всего стиль программирования[7], в Scala выразимый неявными параметрами [32]. Стиль, впрочем, держится на соглашении; встроенная проверка

[6] Неявная передача – распространённый, но не единственный вариант: возможности могут передаваться и явными аргументами или вводиться конструкциями обработчиков.

[7] Функция не обращается к эффекту через глобальное имя или встроенную конструкцию, а – как `business2` – принимает объект-возможность и вызывает его методы.

исключений делает его обязательным: компилятор сам требует возможность `CanThrow` у каждого `throw`.

Главное же новое обстоятельство: возможности могут захватываться функциями первого класса. Так возникает лёгкая альтернатива классическому полиморфизму по эффектам – *контекстный полиморфизм* [13, 33]: функции `map` можно дать привычный тип без какого-либо явного полиморфизма по эффектам – в отличие от строковой `map<e, a, b>` (раздел 4):

```
fun map<a, b>(f: (a) -> b, xs: List<a>): List<b>
```

К сожалению, для возможностей с ограниченным временем жизни обеспечение безопасности требует дополнительных усилий: возможность легко захватывается замыканием, и захватившее значение может пережить предоставивший её контекст – утечь[8]. Например, если свидетельство `CanThrow` покинет пределы блока `try-catch`, гарантия того, что все исключения обработаны, – а с ней и безопасность эффектов (раздел 2) – будет потеряна:

```
fun unsafe() {
    var f
    try { // доступна возможность h: CanThrow<SomeException>
        let g = () => throw SomeException() // захватывает h
        f = g // утечка вычисления, захватившего возможность
    } catch (e: SomeException) { ... }
    f() // бросает SomeException вне обработчика
}
```

Чтобы не допустить утечки, нужен escape-анализ, причём обязательно *интерпроцедурный*: судьба захватившего замыкания зависит от того, что сделает с ним вызываемая функция. При модульной проверке это поведение приходится резюмировать на границах функций, поэтому анализ неизбежно оставляет след в сигнатурах: они сообщают, какие аргументы используются лишь по месту, а какие могут пережить вызов.

Так, `map` лишь применяет `f` к элементам, но не встраивает эту функцию в результат, и escape-анализ фиксирует это в сигнатуре, например пометкой `local`:

```
fun map<a, b>(local f: (a) -> b, xs: List<a>): List<b>
```

Именно неутекание и позволяет `f` безопасно захватывать возможности: раз она не покидает вызов `map`, захваченные возможности остаются в предоставившем их контексте[9].

Функция `withDb` из раздела 4 теперь приобретает следующий вид – `f` принимает неявный параметр типа `Db`, предоставляемый `withDb`:

```
fun withDb(local f: context(local Db) () -> Unit): Unit
```

Пометка `local` при возможности `Db` обязательна по существу: `withDb` закрывает соединение по завершении, поэтому от `f` требуется использовать возможность лишь по месту – она не должна пережить вызов ни в захватившем её замыкании, ни в изменяемой ячейке (сценарии `leak` из раздела 2 и `unsafe` выше). Пометка при самом аргументе `f` – вердикт о поведении `withDb`: аргумент используется лишь по месту, что и позволяет `f` свободно захватывать другие возможности из области вызова `withDb`.

В предложенной оптике системы на основе возможностей делают явными *обе* фазы цикла. Фазу ожидания выражает список неявных параметров вычисления – перечень его контекстных требований: он играет роль, близкую к строке эффектов, но специальные строковые типы для записи этой фазы не обязательны. Новой же является поддержка фазы *захвата* – ровно той, которая строковым системам недоступна: полученное значение здесь можно захватить замыканием и нести внутри вычисления. Удержание захваченной

[8] Возможность же со временем жизни всей программы – скажем, глобальный логгер – может свободно покидать вызовы.

[9] Пометка `local` описывает поведение самой `map` по отношению к аргументу – `f` не сохраняется в её результате. О захватах внутри значений, которые `f` возвращает, она не говорит: те отражаются независимо, в типе `b`.

зависимости в области предоставившего её контекста резюмируют в сигнатурах вердикты escape-анализа – такие как пометка `local`.

Отсюда два следствия, объясняющие устройство семейства. Во-первых, контекстный полиморфизм – прямое следствие захвата лексически видимых возможностей: функция высшего порядка не обязана быть полиморфной по эффектам, потому что её аргумент уже несёт в себе свои возможности. Во-вторых, главной трудностью становится не учёт ожидающих переменных, а *escape-анализ*: нужно гарантировать, что значение, захватившее возможность, не переживёт область обработчика или другого контекста, который эту возможность предоставил.

Запрет утечки воплощают по-разному, и подходы расходятся в цене гибкости. Самый грубый – значения второго класса: возможностям запрещено попадать в свободно переносимые значения первого класса – в частности, в утекающие замыкания, результаты и долгоживущие изменяемые ячейки [34]. Гораздо гибче *полиморфизм по временам жизни*: захваченной возможности позволяют попасть даже в результат, если сигнатура указывает, как время жизни результата соотносится с временами жизни аргументов (регионы [35], времена жизни Rust [36], экспериментальные множества захвата Scala [14]). Цена гибкости – в том, что эти времена жизни и множества дополнительно нагружают сигнатуры. Родственные механизмы удержания обсуждаются в разделе 9.

6. Модальные системы эффектов

Системы на основе возможностей позволили указывать в типах лишь собственные требования вычисления, умалчивая о зависимостях, которые его аргументы уже несут захваченными. Однако на утечку они отвечают запретом (раздел 5). Модальные системы тоже стремятся явно указывать как можно меньше частей контекста и опираются на тот же escape-анализ, но на обнаруженную утечку отвечают иначе: не запрещают её, а повторно освобождают захваченную переменную, возвращая её в фазу ожидания.

Ключевая идея модальной записи – помечать *переходы* между контекстами эффектов, а не повторять требования в типе каждой промежуточной функции. Для этого модальная система [15] использует два конструктора модальных типов, параметризуемых метками эффектов; значения этих типов полноправны (могут храниться и передаваться свободно). Вычисления упаковываются с приписанной модальностью, описывающей требуемый контекст, а в подходящем контексте автоматически распаковываются.

Абсолютная модальность (квадратные скобки) описывает весь требуемый контекст. Например, знакомая нам `business`-функция требует эффект `db` из раздела 4, а `map` не накладывает ограничений на контекст:

```
business : [db](1 -> 1)
map      : [](a -> b) -> List a -> List b)
```

Скобки `[db]` у `business` говорят, что для запуска нужен контекст, поддерживающий `db`; пустые скобки `[]` в типе `map` – что тело замкнуто и не зависит от контекста эффектов, поэтому вызывать её можно в любом контексте; тип `1` здесь обозначает единичный тип (unit).

Относительная модальность (угловые скобки) описывает относительные требования: упакованное вычисление может использовать как эффекты окружающего контекста, так и перечисленные в типе. Естественная среда для неё – типы *обработчиков эффектов*: переданное обработчику вычисление выполняется под ним и потому может использовать, сверх эффектов окружающего контекста, ещё и обрабатываемые – ровно то, что и описывает относительная модальность. Так, у знакомой нам функции `withDb` (раздел 4) тип аргумента получает относительную модальность:

```
withDb : <db>(1 -> 1) -> 1
```

Угловые скобки `<db>` означают, что аргумент вправе выполнять `db` *в дополнение* к эффектам окружающего контекста, а сам `withDb` это требование удовлетворяет.

Устроена система на стыке двух понятий: модальностей и *видов* (kinds). Модальность, приписанная упакованному вычислению, выражает его *остаточное неразрешённое* требование к контексту. Система же видов в нашей оптике играет роль escape-анализа в примитивной форме – грубой двоичной классификации типов по возможной зависимости их значений от контекста. Вид *Abs* получают типы, чьи значения заведомо не зависят от окружающего контекста эффектов (таковы вычисления под абсолютной модальностью, свободно переносимые между контекстами), вид *Any* – типы, значения которых могут нести захваченную зависимость [15]. Когда значение, тип которого имеет вид *Any*, возвращается из-под обработчика – или иначе пересекает границу контекста, – система не отвергает программу, а выражает захваченную зависимость в типе относительной модальностью, вновь делая её явной.

Продемонстрируем это на знакомом эффекте. Пусть в области действия обработчика `db` – внутри `withDb` – определена функция `save`, сохраняющая данные с помощью запросов. Пока она используется там же, её тип – обычный `1 -> 1`, а зависимость от `db` остаётся захваченной. Если же `save` покидает область обработчика (например, возвращается как результат), результат переноса получает тип с относительной модальностью – схематически:

```
save        : 1 -> 1        // внутри области обработчика db
escapedSave : <db>(1 -> 1)  // упакованный результат переноса
```

Захваченная переменная вновь стала ожидающей и видимой в типе: применить `escapedSave` теперь можно лишь в контексте, поддерживающем `db`. `escapedSave` – модально *упакованное* значение, чей тип повторно выражает зависимость исходного `save` от контекста.

С позиций нашего прочтения модальная система работает с той же фазой захвата, что и системы на основе возможностей, и по существу отличается лишь ответом на утечку: вместо того чтобы просто запрещать захваченной контекстной переменной покидать контекст, она вновь выставляет её как требование к будущему контексту – проводит по обратному ребру рис. 1. Этот переход – приписывание относительной модальности пересекающему границу значению, как у `save`, – и есть *повторное освобождение* (раздел 3): захваченная переменная возвращается в фазу ожидания, и тип снова отражает её как контекстное требование.

Уточним, что именно здесь захвачено: в отличие от возможностей, где замыкание буквально держит значение-свидетельство, модальная система не обязана материализовать свидетельство – значение просто остаётся зависимым от контекста, разрешившего его требование. Потому повторное освобождение есть типовая переабстракция этой зависимости, а не извлечение обработчика из замыкания во время выполнения. В исчислении Met [15] этому прочтению отвечает конкретный механизм: значение, возвращаемое обрабатываемым вычислением, упаковывается относительной модальностью обрабатываемых эффектов правилами самой системы – типизации обработчика (T-Handler) и возврата из-под него (E-Ret). Термин фиксирует предлагаемую здесь абстракцию над механизмом модальных типов, а не новое правило поверх системы.

Таким образом, здесь контроль утечки не сводится к *запрету*, как в разделе 5: часть утечек превращается в повторное выставление контекстной переменной в типе. Подчеркнём, что предотвращение небезопасных утечек никуда не исчезает: его выполняет та же классификация видами.

7. Сравнительный анализ

Соберём три семейства в единую картину. Предложенная точка зрения сводит сравнение семейств – каждого в его характерном облике – к двум вопросам о жизненном цикле контекстной свободной переменной: может ли разрешённая зависимость сохраниться внутри значения – буквально, как захваченное свидетельство, либо в широком смысле, как зависимость от контекста построения; а если может – что происходит при переносе такого значения за границу контекста. Строковые системы отвечают «нет» на первый вопрос: разрешённая зависимость не сохраняется в значении, и в типе отложенного вычисления

переменная остаётся ожидающей. Системы на основе возможностей и модальные отвечают «да» и расходятся во втором: первые запрещают утечку, вторые возвращают утёкшую переменную в фазу ожидания. Всё остальное – сама запись фазы ожидания, какой требуется полиморфизм и, в особенности, где сосредоточиваются аннотации – оказывается *следствием* этих ответов, а не независимым измерением. Сравнение приведено в табл. 1.

Табл. 1. Три семейства систем эффектов вдоль единой оси жизненного цикла.
Table 1. Three families of effect systems along the single lifecycle axis.

	Строковые	**Возможности**	**Модальные**
Явные фазы и переходы	ожидание (строка меток)	ожидание (неявные параметры), захват и удержание (вердикты escape-анализа)	ожидание (модальности), захват и контроль переноса (escape-анализ), повторное освобождение (относительная модальность)
Сохранима ли разрешённая зависимость в значении?	нет: в типе требование остаётся неразрешённым	да: буквальным захватом	да: зависимостью от контекста построения
Полиморфизм (следствие)	по строкам	контекстный (через захват); при тонком escape-анализе – по временам жизни	по объемлющему контексту; переменные эффектов – в расширении Met
Аннотации (следствие)	в сигнатурах, через которые распространяется требование	в объявлениях неявных параметров и вердиктах escape-анализа	в точках переноса между контекстами
Обеспечение безопасности	захват невыразим; запуск – лишь в контексте, поддерживающем все эффекты из строки	запрет утечки, обнаруженной escape-анализом	распаковка – лишь в подходящем контексте; утёкшая переменная возвращается в ожидание
Цена	повсеместный полиморфизм по строкам	escape-анализ	модальности в точках переноса; escape-анализ
Примеры систем	Koka, Links, проверяемые исключения Java	Effekt, Scala	Met [15]

Закономерность видна в строках-следствиях табл. 1: выбор фаз и переходов определяет, какой полиморфизм *необходим* дисциплине и где сосредоточиваются аннотации. Обоснование складывается в короткую цепочку следствий. Если разрешённая зависимость не может скрыться внутри значения, требование остаётся неразрешённым и вынуждено распространяться через типы вызывающих вычислений: отсюда специальные row-типы и повсеместный полиморфизм по строкам у функций высшего порядка. Если же зависимость может сохраниться в значении – буквальным захватом свидетельства или в широком смысле, – требование прячется внутри него. Это даёт контекстный полиморфизм и избавляет от явных

переменных эффектов (фазу ожидания записывает сам список неявных параметров или модальность), но делает необходимым escape-анализ – контроль переноса такого значения. Сам анализ может быть устроен грубее или тоньше – от значений второго класса до полиморфизма по временам жизни; дисциплина же, в свою очередь, выбирает ответ на обнаруженную утечку: удержание либо повторное выставление зависимости в типе как требования к принимающему контексту – повторное освобождение. Поэтому вид полиморфизма и основные места аннотаций следуют из способа представления зависимости, точности контроля переноса и реакции на утечку. Захват при этом не запрещает полиморфизм по наборам ожидающих переменных – он делает его необязательным.

За разнообразием полиморфизмов, в свою очередь, стоит общий механизм: во всех трёх семействах полиморфизм – способ записать в сигнатуре зависимость вызова и его результата от аргументов, и различается лишь то, *что* обозначает переменная. В строках она обозначает множество меток – имён неразрешённых требований; в полиморфизме по временам жизни – времена жизни захваченных возможностей или, как в множествах захвата Scala, сами эти возможности. Модальной же записи в характерном случае явная переменная не нужна – её роль играет сам объемлющий контекст; там же, где полиморфизм по эффектам всё-таки нужен явно, его обеспечивает расширение Met переменными эффектов [15].

Подчеркнём: жизненный цикл – аналитическая ось, а не разбиение систем на непересекающиеся классы. Реальные дисциплины сочетают приёмы: есть строковые системы с лексическим связыванием обработчиков (раздел 4), передача свидетельств материализует строковое требование значением, ZIO [37] совмещает свидетельства с полиморфизмом по наборам требований, а упаковка `box` языка Effekt отвечает модальной записи (раздел 9). Таблица описывает характерный облик каждого семейства, а не его границы.

Конкретность различия удобно увидеть на сквозном примере `withDb` (табл. 2; сигнатуры повторяют разделы 4-6). В строковой системе тип называет всё: обрабатываемую метку `db` в строке аргумента и переменную `e` для остальных его эффектов, которую `withDb` повторяет в собственной строке эффектов, делегируя их внешнему контексту. В системе на основе возможностей обрабатываемый эффект стал неявным параметром типа `Db`, а об остальных эффектах `f` тип умалчивает: они приходят захваченными возможностями – контекстный полиморфизм. Умолчание оплачено пометками `local` – вердиктами escape-анализа прямо в сигнатуре: и сам аргумент, и выданная ему возможность используются лишь по месту. В модальной системе обрабатываемый эффект выражен относительной модальностью `<db>`, об остальных же эффектах тип умалчивает и здесь: аргумент вправе использовать эффекты объемлющего контекста [15]. Не нужны здесь ни переменные эффектов – их роль играет объемлющий контекст, – ни пометка `local`: допустимые контексты использования значения уже заданы его относительной модальностью, а попав в подходящий – под обработчик в `withDb` – оно просто распаковывается.

Табл. 2. Тип `withDb` в трёх семействах
Table 2. The type of `withDb` across the three families

Семейство	**Тип `withDb`**
Строковая	`fun withDb<e>(f: () -> Unit / {db | e}): Unit / {e}`
Возможности	`fun withDb(local f: context(local Db) () -> Unit): Unit`
Модальная	`withDb : <db>(1 -> 1) -> 1`

8. Синтез: система эффектов из двух знакомых частей

Единая ось подсказывает конструктивную программу: статическую дисциплину системы эффектов, понимаемой как управление контекстными свободными переменными, можно проектировать как сочетание двух независимо полезных и уже знакомых практикам

механизмов – *неявных параметров* (для фазы ожидания) и *escape-анализа на уровне типов* (для фазы захвата). Пара не задаёт семантику самих эффектных операций и обработчиков – она даёт механизм их статического контроля. Рецепт повторяет жизненный цикл (рис. 1):

1. неразрешённые требования объявляются неявными параметрами: функция `business` из раздела 5 не открывает соединение сама и не тянет его лишним аргументом, а объявляет `context(db: Db)`;
2. связывание выполняет место вызова, предоставляя значение через `context let`;
3. замыкание может захватить полученное значение – это даёт контекстный полиморфизм, – а escape-анализ удерживает захваченное в области контекста, и его вердикт виден прямо в сигнатуре (пометки `local` в табл. 2);
4. при попытке переноса за границу контекста система либо запрещает его (раздел 5), либо выполняет повторное освобождение (раздел 6); заметим, что и во втором случае для определения места, где освобождение нужно произвести, требуется та или иная форма escape-анализа.

В языке, уже располагающем неявными параметрами, такой путь может потребовать меньше новых конструкций, чем полноценные row-типы: реализации неявных параметров уже существуют [31, 38-39]. Польза escape-анализа на уровне типов, в свою очередь, выходит за пределы систем эффектов: он позволяет интерпроцедурно сокращать число размещений в куче [40], поддерживать такие языковые механизмы, как заимствование при управлении ресурсами на основе владения [36, 40] и нелокальные возвраты из неутекающих лямбд [41].

9. Смежные работы

Предложенная оптика – не единственная попытка охватить строки, возможности и модальные типы общим взглядом. Ниже мы соотносим её с другими объединяющими подходами и обозначаем историческую линию.

Ближе всего к нам стоят Танг и Линдли [42]. Они вводят ядро Met(X) – обобщение модальной системы [15], параметризованное структурой эффектов, – и кодируют в него как строковую систему Koka (System F^{ε}), так и систему возможностей Effekt (System C), доказывая сохранение типов и семантики. Тем самым они формально подтверждают, что и строки, и возможности выразимы как модальные эффекты, и по ходу дела приходят к наблюдениям, близким нашим: возможность выступает как неявная форма параметрического полиморфизма, а конструкция `box` языка Effekt отвечает абсолютной модальности. Различаются ось и назначение. Их ось – прикреплено ли отслеживание эффектов к типам функций: строковая запись и запись через возможности размещают эффекты на функциональных стрелках, модальная – отделяет их от стрелок, вынося в переходы между контекстами; исходные системы кодируются в один привилегированный целевой язык – модальные типы. Наша ось – *жизненный цикл* контекстной свободной переменной, нейтральный к реализации и не выделяющий ни одного семейства как подложку; модальные типы у нас лишь третье семейство. Отсюда и разное покрытие: мы сознательно не приводим формальных кодирований, которые дают они, зато выделяем отдельным измерением фазу *захвата* с escape-анализом и повторным освобождением, а также охватываем механизмы, остающиеся за рамками их кодирований (например, времена жизни и множества захвата). Там, где Met(X) выражает повторное освобождение механически относительной модальностью, мы читаем его как переход цикла, общего и для систем на основе возможностей. В этом смысле работы дополняют друг друга: формальная реализация единства – у них, объединяющая оптика для проектировщика – у нас. Сама модальная линия при этом восходит к контекстной модальной теории типов [43] и её применению к алгебраическим эффектам [44].

Промежуточное положение занимает язык Frank с его *способностями* и *корректировками* (abilities and adjustments) [45-46][10]: способность описывает весь доступный контекст эффектов, как абсолютная модальность, а корректировка – его изменение при переходе под обработчик, как относительная. Однако у Frank это синтаксическая дисциплина поверх строкового полиморфизма – переменные эффектов вставляет за программиста компилятор, и в нашей оптике его ожидающие переменные остаются строковыми; в Met же модальности – полноправные конструкторы типов, не опирающиеся на скрытые переменные эффектов [15].

Отдельная линия работ посвящена ребру *удержания*: как не дать локальному свидетельству покинуть предоставивший его контекст. Помимо разобранных в разделе 5 значений второго класса и полиморфизма по временам жизни, сюда относятся удержание генеративного токена состояния `runST` полиморфизмом высокого ранга [47], родственная техника для имён обработчиков первого класса [48], scoped capabilities [49] и компиляция возможностей в регионы [50]. В терминах цикла все эти механизмы реализуют один и тот же переход, различаясь точностью escape-анализа и местом появления аннотаций. Носителем вердикта не обязаны быть даже типы: контракты Kotlin [41] фиксируют неутекание лямбды вне типов. Более полное обсуждение можно найти в нашей предыдущей статье [17], а типовой escape-анализ с экзистенциальными временами жизни мы развиваем в [51].

Иную ось объединения задаёт *градуировка*: аннотации эффектов – и их контекстного двойника, коэффектов, – наделяются алгебраической структурой (упорядоченный моноид или полукольцо). Наиболее полно эффекты и коэффекты сведены вместе Габоарди и др. [52], и эту связь мы отмечали в разделе 3. Эта ось – об алгебре аннотаций и их последовательной композиции, тогда как наша – о фазах жизни контекстной переменной; захват, escape-анализ и повторное освобождение остаются вне оси градуировки, что и делает наш слой захвата ортогональным вкладом. Ещё раньше обобщить дисциплину эффектов пытались Марино и Милстейн [53]: их *generic type-and-effect system* параметризуется абстрактной множествоподобной алгеброй привилегий и правилами проверки, но, как и градуированные монады, описывает лишь фазу ожидания, не затрагивая захвата и удержания.

Наконец, стремление к *общей* теории эффектов не ново. Уадлер и Тиманн [54] установили соответствие между системами типов и эффектов (восходящими к [9-10]) и монадами, явно поставив вопрос об общей теории эффектов; обобщённые и градуированные фреймворки выше суть ответы на него – на алгебраической и семантической осях. Наш вклад лежит на иной, прагматической оси – жизненного цикла контекстной переменной – и потому дополняет, а не заменяет их.

10. Заключение

Мы показали, что статический контроль над эффектами важен для надёжности программ и построения ясных границ абстракций. Главный тезис работы состоит в том, что три, на первый взгляд, несвязанных семейства систем эффектов – строковые, на основе возможностей и модальные – допускают единое прочтение: как механизмы управления контекстными свободными переменными вычисления на уровне типов. Семейства различаются тем, какие переходы жизненного цикла такой переменной – от ожидания к захвату и обратно – они делают выразимыми в типах и что происходит при утечке захваченной переменной (табл. 1).

Этот единый взгляд не только упорядочивает существующие конструкции, но и подсказывает структуру пространства решений. Тем самым он может помочь исследователям и разработчикам языков программирования выбирать решения, отвечающие их задачам, и способствовать внедрению систем эффектов в массовые языки.

[10] Термин «способности» (abilities) во Frank и Unison [26] обозначает строковую запись контекста эффектов и не совпадает с «возможностями» (capabilities) – значениями-свидетельствами из раздела 5.

Естественное продолжение работы – формализация предложенного взгляда: точная формулировка переводов между фазами (в духе передачи свидетельств [30]) и метатеория минимального языка, собранного из неявных параметров и escape-анализа на уровне типов.

Список литературы / References

[1]. Kiselyov O., Sabry A., Swords C. Extensible effects: An alternative to monad transformers, ACM SIGPLAN Notices, 2013, vol. 48, no. 12, pp. 59-70. DOI: 10.1145/2578854.2503791.

[2]. Plotkin G.D., Pretnar M. Handling algebraic effects, Logical methods in computer science, 2013, vol. 9. DOI: 10.2168/LMCS-9(4:23)2013.

[3]. Sivaramakrishnan K.C., Leijen D., Pretnar M., Schrijvers T. Algebraic effect handlers go mainstream (dagstuhl seminar 18172). In Dagstuhl reports, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. DOI: 10.4230/DagRep.8.4.104.

[4]. Phipps-Costin L. et al. Continuing WebAssembly with effect handlers, Proceedings of the ACM on Programming Languages, 2023, vol. 7, no. OOPSLA2, pp. 460-485. DOI: 10.1145/3622814.

[5]. Leijen D. Structured asynchrony with algebraic effects. In Proceedings of the 2nd ACM SIGPLAN international workshop on type-driven development, 2017, pp. 16-29. DOI: 10.1145/3122975.3122977.

[6]. Liang S., Hudak P., Jones M. Monad transformers and modular interpreters. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, 1995, pp. 333-343. DOI: 10.1145/199448.199528.

[7]. Schrijvers T., Piróg M., Wu N., Jaskelioff M. Monad transformers and modular algebraic effects: What binds them together. In Proceedings of the 12th ACM SIGPLAN international symposium on haskell, 2019, pp. 98-113. DOI: 10.1145/3331545.3342595.

[8]. Berg B. van den, Schrijvers T. A framework for higher-order effects & handlers, Science of Computer Programming, 2024, vol. 234, p. 103086. DOI: 10.1016/j.scico.2024.103086.

[9]. Lucassen J.M., Gifford D.K. Polymorphic effect systems. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages, 1988, pp. 47-57. DOI: 10.1145/73560.73564.

[10]. Talpin J.-P., Jouvelot P. The type and effect discipline, Information and Computation, 1994, vol. 111, no. 2, pp. 245-296. DOI: 10.1006/inco.1994.1046.

[11]. Sivaramakrishnan K.C., Dolan S., White L., Kelly T., Jaffer S., Madhavapeddy A. Retrofitting effect handlers onto OCaml. In PLDI '21: 42nd ACM SIGPLAN international conference on programming language design and implementation, ACM, 2021, pp. 206-221. DOI: 10.1145/3453483.3454039.

[12]. Leijen D. Koka: Programming with row polymorphic effect types. In Proceedings of the 5th workshop on mathematically structured functional programming (MSFP 2014), 2014, pp. 100-126. DOI: 10.4204/EPTCS.153.8.

[13]. Brachthäuser J.I., Schuster P., Lee E., Boruch-Gruszecki A. Effects, capabilities, and boxes: From scope-based reasoning to type-based reasoning and back, Proceedings of the ACM on Programming Languages, 2022, vol. 6, no. OOPSLA1, pp. 1-30. DOI: 10.1145/3527320.

[14]. Boruch-Gruszecki A., Odersky M., Lee E., Lhoták O., Brachthäuser J. Capturing types, ACM Transactions on Programming Languages and Systems, 2023, vol. 45, no. 4, pp. 1-52. DOI: 10.1145/3618003.

[15]. Tang W., White L., Dolan S., Hillerström D., Lindley S., Lorenzen A. Modal effect types, Proceedings of the ACM on Programming Languages, 2025, vol. 9, no. OOPSLA1, pp. 1130-1157. DOI: 10.1145/3720476.

[16]. Odersky M., Boruch-Gruszecki A., Brachthäuser J.I., Lee E., Lhoták O. Safer exceptions for scala. In Proceedings of the 12th ACM SIGPLAN international symposium on scala, 2021, pp. 1-11. DOI: 10.1145/3486610.3486893.

[17]. Stoyan A. Modern effect systems overview. In 2024 Ivannikov ISPRAS OPEN conference (ISPRAS), IEEE, 2024, pp. 1-7. DOI: 10.1109/ISPRAS64596.2024.10899144.

[18]. Park Y.G., Goldberg B. Escape analysis on lists. In Proceedings of the ACM SIGPLAN 1992 conference on programming language design and implementation, 1992, pp. 116-127. DOI: 10.1145/143095.143125.

[19]. Hannan J. A type-based escape analysis for functional languages, Journal of functional programming, 1998, vol. 8, no. 3, pp. 239-273. DOI: 10.1017/S0956796898003025.

[20]. Petricek T., Orchard D., Mycroft A. Coeffects: A calculus of context-dependent computation. In Proceedings of the 19th ACM SIGPLAN international conference on functional programming (ICFP '14), ACM, 2014, pp. 123-135. DOI: 10.1145/2628136.2628160.

[21]. Gaster B.R., Jones M.P. A polymorphic type system for extensible records and variants, Department of Computer Science, University of Nottingham, 1996. Available at: https://web.cecs.pdx.edu/~mpj/pubs/96-3.pdf, accessed 25.07.2026.

[22]. Gosling J., Joy B., Steele G., Bracha G. Java Language and Virtual Machine Specifications, Addison-Wesley Professional, 2000. Available at: https://docs.oracle.com/javase/specs/, accessed 25.07.2026.

[23]. Venners B., Eckel B., Hejlsberg A. The trouble with checked exceptions, 2003. Available at: https://www.artima.com/articles/the-trouble-with-checked-exceptions, accessed 27.07.2026.

[24]. Hillerström D., Lindley S. Liberating effects with rows and handlers. In Proceedings of the 1st international workshop on type-driven development, 2016, pp. 15-27. DOI: 10.1145/2976022.2976033.

[25]. Madsen M., Pol J. van de Polymorphic types and effects with Boolean unification, Proceedings of the ACM on Programming Languages, 2020, vol. 4, no. OOPSLA, pp. 1-29. DOI: 10.1145/3428222.

[26]. Unison Computing Abilities in Unison, 2024. Available at: https://www.unison-lang.org/docs/fundamentals/abilities/, accessed 27.07.2026.

[27]. Biernacki D., Piróg M., Polesiuk P., Sieczkowski F. Binders by day, labels by night: Effect instances via lexically scoped handlers, Proceedings of the ACM on Programming Languages, 2020, vol. 4, no. POPL, pp. 1-29. DOI: 10.1145/3371116.

[28]. Zhang Y., Salvaneschi G., Beightol Q., Liskov B., Myers A.C. Accepting blame for safe tunneled exceptions, ACM SIGPLAN Notices, 2016, vol. 51, no. 6, pp. 281-295. DOI: 10.1145/2980983.2908086.

[29]. Zhang Y., Salvaneschi G., Myers A.C. Handling bidirectional control flow, Proceedings of the ACM on Programming Languages, 2020, vol. 4, no. OOPSLA, pp. 1-30. DOI: 10.1145/3428207.

[30]. Xie N., Leijen D. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c, Proceedings of the ACM on Programming Languages, 2021, vol. 5, no. ICFP, pp. 1-30. DOI: 10.1145/3473576.

[31]. Lewis J.R., Launchbury J., Meijer E., Shields M.B. Implicit parameters: Dynamic scoping with static types. In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages, 2000, pp. 108-118. DOI: 10.1145/325694.325708.

[32]. Odersky M. et al. The scala language specification, Lausanne, Switzerland, 2004. Available at: https://www.scala-lang.org/files/archive/spec/, accessed 25.07.2026.

[33]. Brachthäuser J.I., Schuster P., Ostermann K. Effects as capabilities: Effect handlers and lightweight effect polymorphism, Proceedings of the ACM on Programming Languages, 2020, vol. 4, no. OOPSLA, pp. 1-30. DOI: 10.1145/3428194.

[34]. Osvald L., Essertel G., Wu X., Alayón L.I.G., Rompf T. Gentrification gone too far? Affordable 2nd-class values for fun and (co-) effect, ACM SIGPLAN Notices, 2016, vol. 51, no. 10, pp. 234-251. DOI: 10.1145/3022671.2984009.

[35]. Tofte M., Talpin J.-P. Region-based memory management, Information and Computation, 1997, vol. 132, no. 2, pp. 109-176. DOI: 10.1006/inco.1996.2613.

[36]. Matsakis N.D., Klock F.S. The Rust language. In Proceedings of the 2014 ACM SIGAda annual conference on high integrity language technology, 2014, pp. 103-104. DOI: 10.1145/2663171.2663188.

[37]. The ZIO contributors ZIO: Type-safe, composable asynchronous and concurrent programming for Scala, 2024. Available at: https://zio.dev/, accessed 27.07.2026.

[38]. Serrano A., Akhin M., Bobko N., Gorbunov I., Zarechenskii M., Zharkov D. Context parameters (KEEP-367), 2024. Available at: https://github.com/Kotlin/KEEP/blob/master/proposals/context-parameters.md, accessed 25.07.2026.

[39]. Oliveira B.C., Moors A., Odersky M. Type classes as objects and implicits, ACM Sigplan Notices, 2010, vol. 45, no. 10, pp. 341-360. DOI: 10.1145/1932682.1869489.

[40]. Lorenzen A., White L., Dolan S., Eisenberg R.A., Lindley S. Oxidizing OCaml with modal memory management, Proceedings of the ACM on Programming Languages, 2024, vol. 8, no. ICFP, pp. 485-514. DOI: 10.1145/3674642.

[41]. Akhin M., Belyaev M. Kotlin language specification, Kotlin Language Specification, 2021. Available at: https://kotlinlang.org/spec/, accessed 25.07.2026.

[42]. Tang W., Lindley S. Rows and capabilities as modal effects, Proc. ACM Program. Lang., 2026, vol. 10, no. POPL, pp. 923-950. DOI: 10.1145/3776674.

[43]. Nanevski A., Pfenning F., Pientka B. Contextual modal type theory, ACM Transactions on Computational Logic, 2008, vol. 9, no. 3, pp. 1-49. DOI: 10.1145/1352582.1352591.

[44]. Zyuzin N., Nanevski A. Contextual modal types for algebraic effects and handlers, Proceedings of the ACM on Programming Languages, 2021, vol. 5, no. ICFP, pp. 1-29. DOI: 10.1145/3473580.

[45]. Lindley S., McBride C., McLaughlin C. Do be do be do. In Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages, 2017, pp. 500-514. DOI: 10.1145/3009837.3009897.

[46]. Convent L., Lindley S., McBride C., McLaughlin C. Doo bee doo bee doo, Journal of Functional Programming, 2020, vol. 30, p. e9. DOI: 10.1017/S0956796820000039.
[47]. Launchbury J., Peyton Jones S.L. State in Haskell, Lisp and symbolic computation, 1995, vol. 8, no. 4, pp. 293-341. DOI: 10.1007/BF01018827.
[48]. Xie N., Cong Y., Ikemori K., Leijen D. First-class names for effect handlers, Proceedings of the ACM on Programming Languages, 2022, vol. 6, no. OOPSLA2, pp. 30-59. DOI: 10.1145/3563289.
[49]. Odersky M., Boruch-Gruszecki A., Lee E., Brachthäuser J., Lhoták O. Scoped capabilities for polymorphic effects, arXiv preprint arXiv:2207.03402, 2022. DOI: 10.48550/arXiv.2207.03402.
[50]. Müller M., Schuster P., Starup J.L., Ostermann K., Brachthäuser J.I. From capabilities to regions: Enabling efficient compilation of lexical effect handlers, Proceedings of the ACM on Programming Languages, 2023, vol. 7, no. OOPSLA2, pp. 941-970. DOI: 10.1145/3622831.
[51]. Stoyan A.S. Type-based escape analysis with existential lifetimes, Trudy ISP RAN / Proc. ISP RAS, 2026, vol. 38, no. 1, pp. 71-78. DOI: 10.15514/ISPRAS-2026-38(1)-6.
[52]. Gaboardi M., Katsumata S., Orchard D., Breuvart F., Uustalu T. Combining effects and coeffects via grading. In Proceedings of the 21st ACM SIGPLAN international conference on functional programming (ICFP '16), Association for Computing Machinery, New York, NY, USA, 2016, pp. 476-489. DOI: 10.1145/2951913.2951939.
[53]. Marino D., Millstein T. A generic type-and-effect system. In Proceedings of the 4th international workshop on types in language design and implementation (TLDI '09), Association for Computing Machinery, New York, NY, USA, 2009, pp. 39-50. DOI: 10.1145/1481861.1481868.
[54]. Wadler P., Thiemann P. The marriage of effects and monads, ACM Transactions on Computational Logic (TOCL), 2003, vol. 4, no. 1, pp. 1-32. DOI: 10.1145/601775.601776.

Информация об авторах / Information about authors

Андрей Сергеевич СТОЯН – аспирант департамента информатики Национального исследовательского университета «Высшая школа экономики» (Санкт-Петербург). Сфера научных интересов: проектирование языков программирования, системы типов, системы эффектов.

Andrey Sergeevich STOYAN – postgraduate student at the Department of Informatics, National Research University Higher School of Economics (Saint Petersburg). Research interests: programming language design, type systems, effect systems.