УДК 519.685+519.682.2+004.032.22

# Подход к построению системы детерминированного параллельного программирования на основе монотонных объектов

А. И. Адамович, А. В. Климов

В связи с взрывным ростом сложности программ для многоядерных процессоров и суперкомпьютеров в последние десятилетия приобретает популярность и становится всё более актуальной идея параллельных вычислений с детерминированностью, гарантированной языком и системой программирования. В статье анализируется проблема, как сделать параллельное программирование как можно более детерминированным, а также некоторые существующие подходы к её решению. Описываются принципы построения системы объектно-ориентированного программирования, разрабатываемой авторами, предоставляющей возможность писать как детерминированный, так и недетерминированный код с гарантиями прикладному программисту, что его программа будет детерминированной. Система и входной язык имеют два уровня: верхний – для пользователей, разрабатывающих прикладные программы; нижний – для разработчиков библиотек классов, называемых *монотонными*. Входной язык подсистемы верхнего уровня похож на функциональный язык с возможностью создания и использования неизменяемых и монотонных объектов. Библиотеки монотонных классов гарантируют, что все программы на подъязыке верхнего уровня, использующие только монотонные классы, являются детерминированными и идемпотентными при их распараллеливании асинхронными вызовами всех функций. Обсуждаются показательные задачи, реализуемые на данной системе.

*Ключевые слова*: модели параллельных вычислений, детерминированные программы, функциональное программирование, объектно-ориентированное программирование, монотонные объекты.

## 1. Введение

Параллельные и конкурентные (*англ*. concurrent) программы в общем случае являются недетерминированными – дают разные результаты при нескольких прогонах, так как для эффективной реализации на современной аппаратуре требуется явное использование таких средств программирования, как процессы, потоки, треды (*англ*. threads), читающие и изменяющие общие ресурсы и дающие разные результаты при различном порядке доступа потоков к ресурсам. Отладка, модификация и сопровождение таких программ намного более трудоемки, чем привычные, для массовых программистов детерминированных последовательных программ. Поэтому многие языки высокого уровня «прячут» от «обычного» программиста средства конкурентного программирования на уровень реализации и в библиотеки, разрабатываемые экспертами. Однако такие решения ограничивают изобразимые на этих языках классы программ и вынуждают писать менее эффективные приложения, не масштабируемые при увеличении числа процессоров, ядер и других аппаратных средств.

Таким образом, мы имеем следующую ситуацию. Во-первых, детерминированные языки параллельного и конкурентного программирования существуют и развиваются (в качестве

яркого примера приведем чисто функциональный язык Haskell и его библиотеки для параллельного программирования [20]). Во-вторых, нет и не может быть одного языка или библиотеки детерминированного параллельного программирования, который удовлетворил бы все потребности, даже если зафиксировать круг языковых понятий конкурентного программирования, например, объектно-ориентированные языки типа Java с понятием тредов. В результате значительно разнесены уровень детерминированного программирования, считающийся «высоким», и уровень реализации языков и библиотек, считающийся «низким». На этих уровнях используются сильно различающиеся языки и инструменты и требуется очень разная квалификация разработчиков – столь же отличающиеся, как, например, у разработчиков систем программирования и их пользователей.

Возникает вопрос: насколько можно приблизить эти уровни? Нельзя ли в рамках одного языка программирования (пусть это будет, скажем, объектно-ориентированный язык типа Java или Kotlin [9]) дать и универсальные средства недетерминированного программирования для создания базовых инструментов и библиотек, и определить «высокий» уровень в виде подъязыка, проверяемого компилятором, при программировании на котором гарантируется детерминированность. «Нижний», универсальный, уровень потребуется для постоянного расширения и развития предметно-ориентированных библиотек для реализации определенных типов параллельных алгоритмов. Для разработки таких библиотек требуются повышенные трудозатраты, но потом библиотеки используются в большом числе прикладных программах данного класса, программирование, отладка и сопровождение которых становятся намного легче и дешевле благодаря детерминированности, гарантированной библиотеками и системой программирования.

Авторы данной статьи ставят целью дать положительный ответ на этот вопрос в виде двухуровневой системы программирования на языке типа Java. Нижний, базовый, уровень – это весь язык Java, Kotlin [9] или другой, на котором определяются классы, используемые на верхнем уровне. Верхний уровень – это Java-подобный язык или подмножество языка Java, близкое к чисто функциональному языку с естественной (для таких языков) параллельной реализацией и с объектно-ориентированными расширениями, обеспечивающими использование классов нижнего уровня, не нарушая детерминированности параллельных вычислений. Кроме того, мы требуем выполнения еще одного свойства программ – *идемпотентности,* предоставляющего возможность повторного вычисления любого выражения. Библиотечные классы нижнего уровня и их объекты, удовлетворяющие этим требованиям, мы называем *монотонными*. Данный проект продолжает наши работы по Т-системе и монотонным объектам [1–6, 8, 10, 15]. Эта статья является расширенной версией доклада [6].

Основные результаты данной статьи следующие:

- утверждается и демонстрируется на примерах возможность построения открытой системы параллельного программирования на базе объектно-ориентированных языков, в которой прикладному программисту представляется подмножество входного языка, гарантирующее детерминированность и идемпотентность всех его программ при использовании библиотек классов, созданных экспертами и называемых *монотонными*;
- дано формальное операционное определение понятия монотонных классов и объектов (продолжая наши предыдущие работы [5, 6, 8, 15]);
- приведен пример на языках Java и Kotlin определения монотонного объекта, реализующего понятие I-структуры [11], и пример вызывающей его программы на языке Kotlin с использованием средств параллельного программирования на сопрограммах;
- кратко охарактеризованы вопросы программирования на монотонных объектах двух классов задач: порождение эффективного объектно-ориентированного представления графов (невозможного на чисто функциональных языках) и оптимизационных переборных алгоритмов типа метода ветвей и границ.

Нам не известны работы по построению аналогичной системы детерминированного и идемпотентного объектно-ориентированного программирования.

Статья имеет следующее содержание. В разделе 2 дается обзор некоторых существующих подходов к детерминированности программ, которые ближе всего перекликаются с нашим подходом. В разделе 3 описывается двухуровневая архитектура системы детерминированного параллельного программирования. Раздел 4 объясняет, зачем к требованию детерминированности мы добавляем требование идемпотентности. В разделе 5 дается определение монотонных классов и объектов – ключевого понятия нашей работы. В разделе 6 приведен пример монотонного объекта, реализующего понятие I-структуры [11], и функции Фибоначчи, запрограммированной с его использованием. Раздел 7 содержит общую характеристику двух классов задач, на которых в настоящее время отрабатывается система и разрабатывается библиотека монотонных классов. Раздел 8 содержит заключение.

## 2. Близкие работы по детерминированному программированию

Тема детерминированности параллельных программ в последние десятилетия становится всё более популярной и актуальной в связи с всё большим распространением параллельных вычислительных устройств: как многоядерных центральных процессоров (CPU, central processing unit), так и разнообразных ускорителей – от графических (GPGPU, general purpose graphics processor unit) до программируемых логических интегральных схем (ПЛИС, FPGA, field-programmable gate array). Параллельное программирование становится всё более массовым, и возникает проблема его удешевления и повышения надежности. В статье [5] дан обзор некоторых методов и средств детерминированного программирования. В этом разделе приведены те из них, которые имеют бóльшее отношение к решаемой нами задаче.

### 2.1. Детерминированный параллелизм чисто функциональных языков

Отправной точкой, общей основой многих языков со средствами параллельного исполнения являются функциональные языки, которые в «чистом» виде не имеют побочных эффектов и потому считаются «легко» распараллеливаемыми. Распараллеливание кода на функциональном языке, конечно, подразумевает сохранение семантики языка, то есть эквивалентность результата параллельного исполнения эталонному последовательному, а также денотационной семантике, и тем самым – детерминированность.

Максимально параллельное исполнение функциональной программы делается вызовами каждой функции в отдельном параллельном процессе. Это отнюдь не является эффективным решением в общем случае: возникает задача, наоборот, ограничивать параллелизм, чтобы эффективно использовать аппаратные ресурсы, то есть не «распараллеливать», а «секвенциализировать» (*англ.* sequentialize), объединять потенциально параллельные группы вычислений в последовательный поток управления, тред. Прагматичное решение (которому мы следуем в нашем проекте) – дать программисту языковые средства обозначать, где параллельные вызовы, а где последовательный код. Если двигаться с другой стороны – от последовательных языков к параллельным, то аналогичное решение наблюдаем, когда вводится конструкция (или библиотечные средства), называемая в ряде языков «future» и «promise»: вызов функции в отдельном процессе, исполняемом параллельно до тех пор, пока вызвавшему процессу не понадобится результат и он не встанет на ожидание завершения вызванной функции.

Большинство языков, называемых функциональными, – «грязные», то есть разрешают побочные эффекты и не прячут понятие параллельного процесса, треда, позволяя явно управлять ими так же, как и в объектно-ориентированных языках со средствами параллелиз-

ма. Из распространенных функциональных языков лишь Haskell старается сохранять «чистоту» и осторожно расширяется средствами параллельных вычислений [20].

Среди большого разнообразия публикаций по этому направлению особенно интересен сборник, подводящий итоги по состоянию на конец 1990-х годов [14]. Не выходя за рамки функциональной модели вычислений, также строятся проблемно-ориентированные декларативные языки для конкретных областей применения. Например, таким является язык Норма [7] для решения сеточных задач некоторого класса из математической физики.

С другой стороны, современные объектно-ориентированные языки содержат в качестве своего подмножества чисто функциональный язык, на котором можно программировать в функциональном стиле без побочного эффекта. Таковы Java, C#, Kotlin, JavaScript и другие. Здесь основное неудобство – то, что компилятор не проверяет принадлежность к функциональному подмножеству и соблюдение этого условия – дело программиста.

### 2.2. Неизменяемые объекты, immutability

В мире объектно-ориентированных языков есть ряд работ по введению ограничений, проверяемых компилятором и/или во время исполнения и обеспечивающих нужные свойства. Если потребовать, чтобы все объекты были неизменяемыми (*англ.* immutable), то есть их состояние не менялось после отработки инициализаторов, то объектно-ориентированный язык практически превращается в функциональный, которому присущ детерминированный параллелизм.

На практике трудно обходиться совсем без изменений состояния и побочных эффектов, поэтому понятию неизменяемости (*англ.* immutability) придают различную степень «строгости». Статья [16] содержит обзор работ по этой теме.

### 2.3. Статические методы обеспечения детерминированности

Имеется много работ по статическому анализу кода с побочными эффектами, имеющему целью выявлять случаи, когда параллельное исполнение сохраняет детерминированность результата из-за отсутствия «гонок». Здесь особо отметим разработку языка Deterministic Parallel Java, DPJ [12] как имеющую прагматическую цель.

В наших работах эти результаты не используются, мы подходим к задаче с другой стороны: отказываясь от сложного статического анализа полного объектно-ориентированного языка мы накладываем на этот язык синтаксические ограничения, выделяя функциональное подмножество, и реализуем в монотонных классах динамические проверки необходимых по их семантике условий.

### 2.4. Обеспечение детерминированности операциями над данными

Следующий подход к обеспечению детерминированности параллельных программ, в отличие от методов предыдущего раздела, не использует никаких статических средств анализа. За основу берется чисто функциональный язык программирования, быть может, со всевозможными расширениями, не нарушающими функциональность и распараллеливаемость. Затем для взаимодействия параллельных процессов предоставляются специальные структуры данных с операциями, определенными таким образом, чтобы не нарушалась детерминированность. Эта идея породила ряд работ, например:

- I-структуры (*англ.* I-structures) [11];
- сети Кана (*англ.* Kahn networks) [17];
- TStreams, Concurrent Collections [13];
- структуры данных на решетках (*англ.* lattice-based data structures) [18, 19].

У этих подходов есть общая черта: переменные, объекты, через которые осуществляется взаимодействие параллельных процессов, меняют свое состояние монотонно вверх на некоторой полурешетке от неопределенного состояния (⊥) к «всё более определенному». При этом верхний элемент решетки (⊤) обозначает «переопределено», «противоречие»; в программе это соответствует ошибке, выработке исключения. Например, множество значений I-структур с целыми числами описывается решеткой, называемой «плоской», состоящей из нижнего элемента «не определено» (⊥), не сравнимых между собой целых чисел и верхнего элемента «переопределено» (⊤). При выполнении операции присваивания значения $y$ в переменную со значением $x$ в нее записывается наименьшая верхняя грань значений $x$ и $y$. Если полученный результат оказывается верхним элементом ⊤, то вырабатывается исключение.

Эта идея в общем виде была проработана в диссертации Lindsey Kuper [18] и в публикациях вместе с ее коллегами [19]. Она доказала детерминированность параллельных вычислений для процессов, взаимодействующих через переменные, принимающие значения из произвольной (полу)решетки.

В нашем проекте мы используем эти идеи, обобщая их на объекты, определяемые пользователем, с монотонно изменяющимся состоянием.

## 3. Архитектура двухуровневой системы детерминированного параллельного программирования

Поскольку не существует единого набора средств, зафиксированных в языке программирования и библиотеке, обеспечивающих все случаи детерминированного параллельного программирования, система должна быть открытой и входной язык должен позволять как детерминированный, так и недетерминированный код. Но чтобы дать гарантии детерминированности прикладному программисту, входной язык системы должен быть двухуровневым:

- верхний уровень – детерминированная часть: прикладной код на подмножестве языка, который пишет прикладной программист. Ему гарантируется детерминированность любой программы, использующей заготовленные библиотеки нижнего уровня. Принадлежность детерминированному подмножеству проверяется компилятором. Оно примерно соответствует функциональному подмножеству языка Java и ему подобных;
- нижний уровень – недетерминированная часть: библиотеки классов, создаваемые квалифицированными программистами для определенных областей применения на универсальном объектно-ориентированном языке типа Java с богатым набором изобразительных средств, позволяющем кодировать недетерминированные параллельные алгоритмы. Авторы библиотек гарантируют детерминированность параллельных приложений их авторам, программирующим на функциональном подъязыке верхнего уровня.

Такие классы и объекты нижнего уровня мы называем *монотонными*, поскольку оказывается, что, как и в работах, упомянутых в разделе 2.4, их состояние меняется монотонно на некоторой полурешетке, которую, как правило, можно построить по коду класса.

Входным языком такой системы может быть любой объектно-ориентированный язык со средствами параллельного и конкурентного программирования, у которого можно выделить функциональное подмножество. Таковыми являются большинство современных объектно-ориентированных языков. Однако в таком случае некоторые детали эффективной реализации будут «торчать» и загромождать прикладной код. Поэтому мы разрабатываем специализированный Java-подобный язык, названный Ajl [3], в котором эти детали прикрыты синтаксическим сахаром и генерируются из компактного кода. Таким способом мы получаем возможность предлагать пользователям разные механизмы реализации параллельных процессов, тредов, легких тредов (*англ*. light-weight thread).

Мы проводим прототипирование системы, используя платформу JVM[1], языки Java и Kotlin[2] [9], сопрограммы языка Kotlin[3], библиотеку Quasar[4], реализующую легкие потоки («файберы» и «стренды», *англ.* fibers and strand) на JVM. Отслеживаем развитие проекта Loom[5], идущего на смену Quasar с целью более тесной интеграции с языком Java и последующего включения в стандартную Java-библиотеку. Разрабатываем свой язык Ajl [3].

## 4. Роль идемпотентности

Помимо детерминированности есть еще одно важное понятие, которое из узко математического превращается во всё чаще упоминаемое ценное свойство параллельных и распределенных программ: *идемпотентность*[6]. Операция, вызов процедуры или функции называется *идемпотентной*, если ее можно выполнить повторно с копией аргументов и она выдаст эквивалентный результат и не породит нового побочного эффекта или породит лишь такой, какой не будет программно заметен из вызвавшего операцию кода. Идемпотентную операцию можно прервать, не завершив, а потом вызвать повторно, досчитать до конца и использовать результат повторного вызова вместо неполученного результата первого. Повторное вычисление завершит создание побочного эффекта, и ни эта, ни другие подсистемы не заметят, что вычисление прерывалось и повторялось. Многократное выполнение идемпотентной операции эквивалентно однократному.

Очевидно, что идемпотентность особенно важна в распределенных системах с большим количеством узлов, будь то узлы суперкомпьютера или вычислительные установки, взаимодействующие через интернет. В таких задачах и системах ненулевая частота отказов узлов и подсистем должна учитываться при их проектировании. Реализация идемпотентных операций, которые можно перевызывать при отказе или по таймауту, – самый простой и естественный способ обеспечения надежности и безотказности приложений.

Например, в интернет-протоколе HTTP[7] некоторые операции, изменяющие состояние, идемпотентны, а именно, PUT и DELETE, а операции POST и PATCH неидемпотентные. Эта разница оговорена в спецификации протокола и должна учитываться разработчиками и серверов, и приложений. Всякий раз, когда можно применить идемпотентную операцию, следует это делать. При использовании неидемпотентных операций программирование обработчиков исключительных ситуация становится более сложным, чем с идемпотентными.

В чисто функциональных языках все вызовы функций идемпотентны, так как не создают побочного эффекта. Естественно перенести это свойство и на детерминированное объектно-ориентированное программирование с побочными эффектами. Более того, оказывается, что теория и приемы разработки монотонных объектов становятся более ясными, если проектировать объекты нижнего уровня сразу с учетом этого и детерминированными, и идемпотентными. По нашему опыту разработки монотонных классов, легче позаботиться об идемпотентности, после чего детерминированность часто получается сама собой или нужно приложить лишь немного дополнительных усилий, чтобы ее достичь.

[1] JVM = Java Virtual Machine.
[2] `http://kotlin.jetbrains.org/`
[3] `https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html`
[4] `https://github.com/puniverse/quasar`
[5] `https://wiki.openjdk.java.net/display/loom/Main`
[6] `https://en.wikipedia.org/wiki/Idempotence`
[7] `https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol`

## 5. Понятие монотонного класса и объекта

Резюмируем определение монотонности классов и объектов, как оно используется в нашей разработке и было дано в предыдущих работах [15, 8, 6, 10]. Оно дается не через наличие полурешетки, как в [18, 19], а операционно.

Классы и их объекты называются *монотонными*, если при их использовании в любой программе на функциональном подъязыке все выражения удовлетворяют следующим двум свойствам:

- *Детерминированность*: результаты, полученные в любом порядке параллельных вычислений копий выражения от одного и того же начального состояния, эквивалентны.
- *Идемпотентность*: повторное вычисление копии выражения от состояния, полученного при предыдущем вычислении или параллельно с ним, дает эквивалентный результат и побочный эффект, не отличимый программно на подъязыке верхнего уровня от побочного эффекта первого вычисления.

Эти свойства должны выполняться одновременно для всех монотонных классов, когда они используются вместе, в любой функциональной программе. Здесь используется понятие эквивалентности результатов как их неотличимости средствами языка верхнего уровня.

Обратим внимание на следующую тонкость этого определения: монотонность классов и объектов определяется не через их свойства как таковые, а через свойства *любой* функциональной программы, использующей эти классы. Поэтому доказательства монотонности, вообще говоря, нетривиальны.

В настоящее время мы конструируем монотонные классы, рассуждая об этих свойствах неформально и наивно. В будущем мы рассчитываем, что удастся разработать средства автоматизации доказательства монотонности в подавляющем числе случаев, хотя, конечно, полностью автоматических компьютерных инструментов на все случаи жизни никогда построено не будет, как и в общем случае средств верификации программ.

## 6. Пример монотонного класса и его использования

Приведем пример простого монотонного класса, имеющего практическое значение, который был изобретен более 30 лет назад в связи с разработкой архитектур с управлением потоком данных (*англ*. dataflow computers). Речь об I-структурах [11].

*I-структурой* называется переменная целочисленного типа (или другого примитивного, то есть нессылочного) со следующей семантикой операций:

- при создании переменная находится в состоянии «не определено», «не готово»;
- при чтении:
  - если переменная имеет состояние «не готово», процесс, вызвавший эту операцию, переходит в состояние ожидание записи в эту переменную каким-либо процессом и будет продолжен после записи;
  - в противном случае выдается текущее значение переменной;
- при записи значения $x$:
  - если состояние переменной «не готово», $x$ записывается в нее и она переходит в состояние «готово»;
  - если значение переменной было равно $x$, то ничего не делается;
  - в противном случае вырабатывается исключение.

```java
class LongM {

  private boolean unready = true;
  private long value;

  public synchronized long get() {

    if (unready) wait();
    return value;
  }

  public synchronized void set(long x) {

    if (unready) {
      value = x;
      unready = false;
      notifyAll();
    }
    else if (x != value)
      throw new RuntimeException();
  }
}
```

Рис. 1. Монотонный класс `LongM` для хранения значения типа `long` на Java

```kotlin
class LongM {

  private var unready = Suspender()
  private var value: Long = 0;

  suspend fun get(): Long {

     unready?.suspend()
     return value
  }

  fun set(x: Long) {
    val u = unready
    if (u != null) {
      value = x
      unready = null
      u.resumeAll()
    }
    else if (x != value)
      throw Exception()
  }
}
```

Рис. 2. Монотонный класс `LongM` для хранения значения типа `Long` на Kotlin

```kotlin
suspend fun fib(n: Int): Long = coroutineScope {

    val v = LongM()
    val w = LongM()
    fibRec(n, v, w)
    w.get()
}

suspend fun fibRec(n: Int, v: LongM, w: LongM): Unit = coroutineScope {

    if (n <= 1) {
        v.set(1)
        w.set(1)
    }
    else {
        val u = LongM()
        launch { w.set(u.get() + v.get()) }
        fibRec(n - 1, u, v)
    }
}
```

Рис. 3. Пример программы на языке верхнего уровня – функциональном подмножестве языка Kotlin, использующем механизм сопрограмм и их параллельного выполнения: вычисление чисел Фибоначчи `fib(n)` с запоминанием промежуточных значений в монотонных объектах `LongM`

На рис. 1 и 2 изображен код на языках Java и Kotlin класса `LongM` с операциями чтения `get` и записи `set` целых чисел типа `long` с семантикой I-структуры. На рис. 1 класс `LongM` реализован с использованием стандартных средств многопоточного программирования языка Java. Булевская переменная `unready` указывает, определено ли значение поля `value`, в которое записывается аргумент операции `set`.

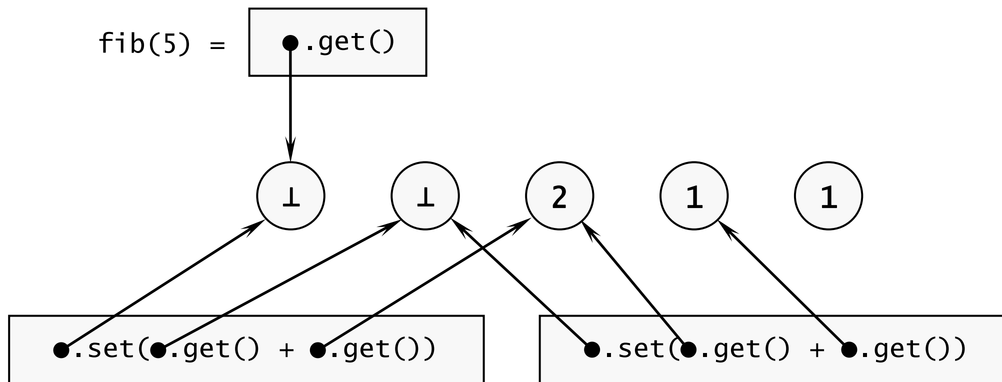


Рис. 4. Состояние вычисления fib(5), после того как рекурсивная функция fibRec отработала, но оставила незавершившиеся сопрограммы, созданные операторами launch. Часть из них выполнилась и заполнила три монотонных объекта LongM значениями 2 = fib(2), 1 = fib(1), 1 = fib(0). Две другие изображены прямоугольниками с выражениями u.set(v.get() + w.get()) со ссылками на объекты на местах переменных u, v, w. Объекты класса LongM изображены кругами, где знак ⊥ обозначает «неготовое» состояние, число – значение поля value.

Рис. 2 содержит код класса с такой же семантикой на языке Kotlin с использованием его средств параллельного программирования на сопрограммах. Методы, помеченные ключевым словом **suspend**, имеют возможность «задерживаться» некоторыми операциями. Стеки вызовов таких методов называются в языке Kotlin *сопрограммами*. Здесь метод get задерживается операцией unready?.suspend() до тех пор, пока переменная value не станет «готовой». Объекты вспомогательного класса Suspender поддерживают очередь задержанных сопрограмм. Метод suspend добавляет текущую сопрограмму в очередь, после чего управление передается диспетчеру, возобновляющему выполнение другой готовой сопрограммы. Метод resumeAll освобождает все сопрограммы в данной очереди, то есть передает их диспетчеру, который их продолжит, когда у него будет такая возможность.

Мы утверждаем (без доказательства), что все программы на функциональном подмножестве языка Java или Kotlin, использующие класс LongM, обладают детерминированностью и идемпотентностью, то есть класс LongM является монотонным.

По сравнению с реализацией класса LongM на Java на «тяжелых» тредах сопрограммы языка Kotlin – это «легкое» средство параллельного программирования. Кроме того, параллельные программы на языке Kotlin более компактны и читабельны, чем на Java. Поэтому пример использования класса LongM приведем на рис. 3 на языке Kotlin. В коде встречаются два идентификатора, относящихся к параллельному программированию: метод coroutineScope, употребление которого воспринимайте как шаблон программирования сопрограмм, запускающих другие сопрограммы, и метод launch, создающий сопрограмму с выражением из аргумента в качестве ее начального состояния. Эта сопрограмма передается диспетчеру и попадает в очередь готовых, а вычисление продолжается после launch. Отметим, что оператор создания объекта записывается в языке Kotlin без ключевого слова **new**.

Программа на рис. 3 вычисляет числа Фибоначчи за линейное время благодаря сохранению промежуточных значений в объектах LongM. Функция fib(n) – головная, fibRec(n, v, w) – вспомогательная. Ее последний аргумент w – монотонный объект класса LongM по окончании вычисления принимает значение числа Фибоначчи fib(n). Рис. 4 иллюстрирует ход вычисления fib(5), показывая некоторое промежуточное состояние сопрограмм и объектов.

Заметим, что код на рис. 4 напоминает программирование на языке Prolog. Это неудивительно: логические переменные языка Prolog – это пример монотонных объектов. Однако

наша задача не ограничивается воспроизведением элементов логического программирования на объектно-ориентированном языке, а направлена на предоставление более универсальных и эффективных средств работы с объектами с явным использованием ссылок, чего нет ни в чисто функциональном, ни в логическом программировании.

## 7. Примеры задач, решаемых с монотонными объектами

Для изучения и демонстрации возможностей программирования с монотонными объектами мы разрабатываем библиотеку монотонных классов на следующих классах задач.

### 7.1. Порождение и обработка графов

Классические чисто функциональные языки не дают возможности обрабатывать графы эффективно, так, чтобы узлы были представлены объектами, а связи между ними – ссылками в полях объектов. В функциональных языках можно эффективно представлять только деревья. В системе программирования с монотонными объектами мы сохраняем большинство положительных свойств функциональных языков и расширяем область эффективно представимых данных с деревьев до произвольных графов.

Однако создание столь же эффективного представления графа средствами ограниченных побочных эффектов, какие допустимы для монотонных объектов, – нетривиальная задача. Посмотрим, что будет, если в классе `LongM` на рис. 1 или 2 заменить тип переменной `value` с примитивного `long` (`Long` соответственно) на ссылочный `Object`, переименовав класс в `ObjectM`, чтобы название отражало смысл. Тогда класс станет немонотонным! Чтобы показать это, рассмотрим такую последовательность предложений (в синтаксисе языка Kotlin):

```
val a: ObjectM = ObjectM()
a.set(ObjectM())            // (1) первое вычисления выражения
a.set(ObjectM())            // (2) второе вычисление такого же выражения
```

Строки (1) и (2) содержат одинаковые выражения, а идемпотентность требует, чтобы выражение (2) вычислялось с таким же результатом, что и выражение (1), и без нового побочного эффекта. Однако после выполнения (1) полю `a.value` будет присвоена ссылка на объект, созданный на строке (1), а вызов `set` в строке (2) попытается записать в то же поле неравную ссылку на новый объект `ObjectM()`, что вызовет выработку исключения.

В работе [10] эта проблема разбирается подробнее и предлагается несколько решений для записи ссылок в монотонные объекты, два из которых позволяют порождать циклические структуры данных, столь же эффективные, как и при обычном, не ограниченном монотонностью, объектно-ориентированном программировании.

Сформулируем идею одного из них, чтобы показать, что описанные трудности преодолимы. Заведем в данном классе, претендующем на монотонность (похожем на `ObjectM` или другом), «фабрику» объектов, которая создает не один объект, а сразу массив или список из указанного числа новых объектов. Этот список не только выдается в качестве результата фабрики, но и сохраняется в приватном поле каждого из созданных объектов. У данного монотонного класса могут быть ссылочные поля. Определим операцию `set` так, чтобы она позволяла записать в эти поля только ссылки из сохраненного списка, то есть на объекты, созданные *одновременно* с данным объектом. При обращении к `set` со ссылкой, отсутствующей в списке, выдается исключение. Можно убедиться, что классы и объекты с такой операцией записи `set` и фабрикой одновременного создания объектов являются монотонными.

Недостаток такого решения: при создании представления графа надо заранее знать число узлов в нем, чтобы одновременно породить нужное число представляющих их объектов. Это «плата» за детерминированность и идемпотентность.

### 7.2. Переборные алгоритмы типа метода ветвей и границ

С монотонными объектами можно запрограммировать, конечно, не все задачи параллельного программирования, а только некоторую часть тех, у которых результат однозначен. Именно таковы оптимизационные задачи типа поиска кратчайшего пути в графе методом ветвей и границ (*англ.* branch-and-bound).

По смыслу задачи в них присутствуют несколько видов монотонно изменяемых данных: монотонно меняется рекорд (кратчайший путь, найденный к настоящему времени), причем порядок его изменения зависит от порядка исполнения параллельных процессов, но результат не зависит; монотонно растут пути, причем их много. Кроме того, необходима эквивалентность реализаций с полным перебором и с «отсевом» (*англ.* pruning) вариантов путей, которые заведомо не улучшат рекорд. Также желательно, чтобы «переключение» реализации с полным перебором на вариант с отсевом делалось в одном месте кода, чтобы упростить доказательство их эквивалентности. Гарантии эквивалентности этих вариантов и монотонности должны быть реализованы в как можно меньшем по размеру коде монотонных классов.

С точки зрения методов разработки монотонных классов эта задача интересна тем, что для реализации «отсева» нужны монотонные объекты «высшего порядка», операции которых принимают в качестве аргумента лямбда-выражение, описывающее продолжения вычислений, которые могут подвергнуться «отсеву».

Эта задача также оказалась нетривиальной и поучительной. Подробная информация о методике программирования таких задач будет опубликована в наших будущих работах.

## 8. Заключение

В статье дана мотивировка и принципы построения системы параллельного объектно-ориентированного программирования, разрабатываемой авторами на основе понятия монотонных классов и объектов. Ее основная идея – двухуровневый входной язык, состоящий из универсального объектно-ориентированного языка «нижнего» уровня (типа Java, Kotlin и т.п.), предназначенного для экспертов в параллельном программировании, создающих библиотеки, и языка «верхнего» уровня – подмножества, близкого к функциональному языку, предназначенного для прикладных программистов. Дано операционное определение понятия монотонности классов и объектов как таких, которые гарантируют использующим их программам на подъязыке верхнего уровня детерминированность и идемпотентность.

Для демонстрации реализуемости и практичности данного подхода приведен пример на языке Kotlin для монотонного класса, реализующего понятие I-структуры, и функции Фибоначчи, его использующей. Пример показывает, что язык Kotlin обладает уже достаточным набором понятий, чтобы развивать на нем методику, которую можно назвать *монотонным программированием*. Однако в общем случае многие понятия реализации будут «торчать» в прикладном коде. Поэтому целесообразна разработка Java/Kotlin-подобного языка, в котором эти элементы реализации «прикрыты» высокоуровневыми понятиями. Кроме того, его компилятор будет проверять принадлежность прикладного кода языку, на котором гарантируется детерминированность и идемпотентность. Мы ведем разработку такого языка с рабочим названием Ajl, и его завершение является нашей будущей работой.

Центральная часть данного проекта – разработка библиотек монотонных классов для различных областей применения, гораздо более сложных, чем приведенный пример. Были охарактеризованы два типа прикладных задач, на которых система сейчас отрабатывается и будет продемонстрирована в будущих публикациях. Не существует конечного набора классов, которые удовлетворили бы все потребности монотонного параллельного программирования, поэтому эта библиотека открыта и должна развиваться и пополняться. Главное предназначение разрабатываемой системы – сделать этот процесс легким и практичным.

## Литература


1. *Абрамов С. М.*, *Адамович А. И.*, *Коваленко М. Р.* Т-система – среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей // Программирование. 1999. Т. 25, № 2. С. 100–107.
2. *Адамович А. И.* Струи как основа реализации понятия Т-процесса для платформы JVM // Программные системы: теория и приложения. 2015. Т. 6, № 4. С. 177–195. DOI: 10.25209/2079-3316-2017-8-4-221-244.
3. *Адамович А. И.* Язык программирования Ajl: автоматическое динамическое распараллеливание для платформы JVM // Программные системы: теория и приложения. 2016. Т. 7, № 4. С. 83–117. DOI: 10.25209/2079-3316-2016-7-4-83-117.
4. *Адамович А. И.*, *Климов А. В.* Об опыте использования среды метапрограммирования Eclipse/TMF для конструирования специализированных языков // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции. М.: ИПМ им. М. В. Келдыша РАН, 2016. С. 3–8. DOI: 10.20948/abrau-2016-45.
5. *Адамович А. И.*, *Климов А. В.* Как создавать параллельные программы, детерминированные по построению? Постановка проблемы и обзор работ // Программные системы: теория и приложения. 2017. Т. 8, № 4. С. 221–244. DOI: 10.25209/2079-3316-2017-8-4-221-244.
6. *Адамович А. И.*, *Климов А.В.* Принципы построения системы детерминированного параллельного программирования // Наукоемкое программное обеспечение: труды семинара 12-й Междунар. Ершовской конф. по информатике (ПСИ'19). 2–3 июля 2019 г., Новосибирск. С. 26–33. ISBN 978-5-4437-0909-3.
7. *Андрианов А. Н.*, *Баранова Т. П.*, *Бугеря А. Б.*, *Ефимкин К. Н.* Трансляция непроцедурного языка Норма для графических процессоров // Препринты ИПМ им. М. В. Келдыша. 2016. № 73. 24 с. DOI: 10.20948/prepr-2016-73.
8. *Климов А. В.* Детерминированные параллельные вычисления с монотонными объектами // Научный сервис в сети Интернет: многоядерный компьютерный мир: труды Всероссийской научной конференции. М.: Изд-во Московского университета, 2007. С. 212–217.
9. *Скин Д., Гринхол Д.* Kotlin. Программирование для профессионалов. СПб.: Питер, 2020.
10. *Adamovich A.I.*, *Klimov A.V.* Building Cyclic Data in a Functional-Like Language Extended with Monotonic Objects // X Workshop PSSV: Program Semantics, Specification and Verification: Theory and Applications. Abstracts. 2019. P. 11-19. ISBN 978-5-4437-0918-5.
11. *Arvind*, *Nikhil R. S.*, *Pingali K. K.* I-structures: Data Structures for Parallel Computing // ACM Trans. Program. Lang. Syst. 1989. V. 11, № 4. P. 598–632. DOI: 10.1145/69558.69562.
12. *Bocchino R. L. (Jr.)*, *Adve V. S.*, *Adve S. V.*, *Snir M.* Parallel Programming Must Be Deterministic by Default // Fifth USENIX Conference on Hot Topics in Parallelism, HotPar'09. USENIX Association, 2009.
13. *Burke M. G.*, *Knobe K.*, *Newton R.*, *Sarkar V.* Concurrent Collections Programming Model // Encyclopedia of Parallel Computing / ed. D. Padua. Springer US, 2011. P. 364–371. DOI: 10.1007/978-0-387-09766-4_238.
14. *Hammond K.*, *Michelson G.* (eds.). Research Directions in Parallel Functional Programming, London, UK: Springer, 1999.
15. *Klimov A. V.* Dynamic Specialization in Extended Functional Language with Monotone Objects // SIGPLAN Not. 1991. V. 26, № 9. P. 199–210. DOI: 10.1145/115865.376287.
16. *Potanin A.*, *Östlund J.*, *Zibin Y.*, *Ernst M. D.* Immutability // Aliasing in Object-Oriented Programming / eds. D. Clarke, J. Noble, T. Wrigstad. LNCS 7850. Springer, 2013. P. 233–269.
17. *Kahn G.* The Semantics of a Simple Language for Parallel Programming // IFIP Congress, 1974. P. 471–475.

18. *Kuper L.* Lattice-based Data Structures for Deterministic Parallel and Distributed Programming, Ph.D. Thesis. IN, USA: Indiana University, 2015. 253 p.
19. *Kuper L.*, *Todd A.*, *Tobin-Hochstadt S.*, *Newton R. R.* Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish // ACM SIGPLAN Not. 2014. V. 49, № 6. P. 2–14. DOI: 10.1145/2666356.2594312.
20. *Marlow S.* Parallel and Concurrent Programming in Haskell. CA, USA: O'Reilly, 2013.

*Статья поступила в редакцию 31.07.2019;*
*переработанный вариант – 02.09.2019.*

**Адамович Алексей Игоревич**
старший научный сотрудник Исследовательского центра мультипроцессорных систем Института программных систем им. А. К. Айламазяна РАН (152021, Ярославская обл., Переславский район, с. Веськово, ул. Петра Первого, д. 4 «а»), тел. (4852) 695-228, e-mail: lexa@adam.botik.ru.

**Климов Андрей Валентинович**
старший научный сотрудник отдела «Программное обеспечение высокопроизводительных вычислительных систем и сетей» Института прикладной математики им. М. В. Келдыша РАН (125047, Москва, Миусская пл., д. 4), тел. (4852) 695-228, e-mail: klimov@keldysh.ru.

## An approach to deterministic parallel programming system construction based on monotonic objects

**A. I. Adamovich, A. V. Klimov**

Due to the explosive growth of the complexity of programs for multi-core processors and supercomputers, the idea of parallel computation with determinism guaranteed by the programming language and system is becoming increasingly significant. This paper analyzes the problem of making parallel programming as deterministic as possible and some of the existing approaches to its solution. It describes the principles of constructing the object-oriented programming system developed by the authors, which allows expert programmers to write both deterministic and non-deterministic code with guarantees to application developers that their programs are deterministic. The system and its input language have two levels: the higher level is intended for application developers; the lower level is for developers of class libraries referred to as *monotonic*. The input language of the higher-level subsystem is like a functional language with the possibility of creating and using immutable and monotonic objects. The libraries of monotonic classes ensure that all programs in the higher-level sublanguage that use only these classes are deterministic and idempotent when they are parallelized by asynchronous calls of all functions. Some representative applications implemented on this system are described.

*Keywords*: models of parallel computation, deterministic programs, functional programming, object-oriented programming, monotonic objects.