УДК 519.685+519.682.2+004.032.22

Подход к построению системы детерминированного параллельного программирования на основе монотонных объектов

А. И. Адамович, Анд. 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 анализируются проблема, как сделать параллельное программирование как можно более детерминированным, и некоторые существующие подходы к ее 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. Приводятся показательные классы задач, 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, сопровождение таких программ намного более трудоемки, чем привычных массовым программистам 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 увеличения числа процессоров, ядер и других аппаратных средств.

Статья подана в журнал «Вестник СибГУТИ», ISSN 1998-6920, 31 июля 2019 г.

Таким образом, мы имеем следующую ситуацию. Во-первых, детерминированные языки параллельного и конкурентного программирования существуют и развиваются (в качестве яркого примера приведем чисто функциональных язык Haskell и его библиотеки для параллельного программирования [19]). 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 (*англ.* threads). 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: на сколько можно приблизить эти уровни? Нельзя ли в рамках одного языка программирования (пусть это будет, скажем, объектно-ориентированный язык типа Java) дать и универсальные средства недетерминированного программирования для создания базовых инструментов и библиотек, и определить «высокий» уровень, подъязык, проверяемый компилятором, при программировании на котором гарантируется детерминированность. «Нижний» универсальный уровень потребуется для постоянного расширения и развития проблемно-ориентированных библиотек для реализации определенных классов алгоритмов. Для разработки таких библиотек требуются повышенные трудозатраты, но потом библиотеки используются во многих прикладных программах данного класса, программирование и отладка которых становится намного легче и дешевле благодаря детерминированности, гарантированной авторами библиотек и системы программирования.

Авторы данной статьи ставят цель дать положительный ответ на этот вопрос в виде двухуровневой системы программирования на языке типа Java. Нижний, базовый уровень – это весь язык Java, на котором определяются классы, используемые на верхнем уровне. Верхний уровень – это Java-подобный язык или подмножество языка Java, напоминающий чисто функциональным язык с естественной (для функциональных языков) параллельной реализацией и с объектно-ориентированными расширениями, обеспечивающими использование классов нижнего уровня, не нарушая детерминированности параллельных вычислений. Библиотечные классы нижнего уровня и их объекты мы называем *монотонными* (по причинам, которые объясним ниже). Проект продолжает наши работы по Т-системе и монотонным объектам [1–6, 8, 9, 14].

Эта работа делает вклад в решение еще одной фундаментальной проблемы: построение модели вычислений, промежуточной между функциональной и объектно-ориентированной, такой что в ней есть явные ссылки на объекты, объекты могут изменяться (хотя и некоторым дисциплинированным образом), но при этом выполняются основные свойства функциональных языков: детерминированность, идемпотентность, наличие денотационной семантики, близкой к классической семантике функциональных языков, и некоторая адекватная замена референциальной прозрачности (*англ.* referential transparency), которая нарушена операциями создания объектов, порождающими новые ссылки, и побочными эффектами.

Статья имеет следующее содержание. В разделе 1 дается обзор некоторых существующих подходов к детерминированности программ, которые ближе всего перекликаются с нашим подходом. В разделе 2 описывается двухуровневая архитектура системы детерминированного параллельного программирования. Раздел 3 объясняет, зачем к требованию детерминированности мы добавляем требование идемпотентности. В разделе 4 дается определение монотонных классов и объектов – ключевого понятия нашей работы. Раздел 5 дает общую характеристику двух классов задач, на которых в настоящее время отрабатывается система и разрабатывается библиотека монотонных классов. Раздел 6 содержит заключение.

Эта статья является расширенной версией конференционной публикации [6].

1. Существующие средства детерминированного параллельного программирования

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 мы даем краткую характеристику тем из них, которые имеют бóльшее отношение к решаемой нами задаче.

1.1 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 каждой программы эталонному последовательному, а также денотационной семантике, 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, ограничить параллелизм, чтобы эффективно использовать аппаратные ресурсы, – то есть не «распараллелить», а «секвенциализировать» (*англ.* sequentialize), объединить 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»[1]: вызов функции в отдельном процессе, исполняемом параллельно до тех пор, пока вызвавшему процессу не понадобится результат и он не встанет на ожидание завершения.

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 параллелизма. Из распространенных функциональных языков лишь Haskell сохраняет «чистоту» и осторожно расширяется средствами параллелизма – как правило, библиотечными, с соответствующей поддержкой на нижнем уровне в реализации языка [19].

Среди большого разнообразия публикаций по этому направлению особенно интересен сборник, подводящий итоги по состоянию на конец 1990-х годов [13]. 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.

[1] https://en.wikipedia.org/wiki/Futures_and_promises

Здесь основное неудобство – это то, что компилятор не проверяет принадлежность функциональному подмножеству и соблюдение этого условия – дело программиста.

1.2 Неизменяемые объекты, immutability

Если двигаться со стороны императивных и объектно-ориентированных языков к функциональным, существует 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 [15] содержит обзор работ по этой теме.

1.3 Статические методы обеспечения детерминированности

Имеется много работ по статическому анализу кода, содержащему побочный эффект, имеющему целью выявить случаи, когда параллельное исполнение сохраняет детерминированность результата из-за отсутствия «гонок». Здесь особо отметим разработку языка Deterministic Parallel Java, DPJ [11] как имеющую прагматическую цель в рамках популярного объектно-ориентированного языка.

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

1.4. 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 так определенными операциями, чтобы не нарушалась детерминированность. Оказывается, это не только возможно, но и такая идея породила ряд направлений, например:

- I-структуры (*англ*. I-structures) [10];
- сети Кана (*англ*. Kahn networks) [16];
- TStreams, Concurrent Collections [12];
- структуры данных на решетках (*англ*. lattice-based data structures) [17, 18].

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 ($\bot$) к «всё более определенному». При этом верхний элемент решетки ($\top$) обозначает «переопределено»; 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» ($\bot$), не сравнимых между собой целых чисел и верхнего элемента «переопределено» ($\top$). 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 $\top$, то вырабатывается исключение.

Эта идея в общем виде была проработана в диссертации Lindsey Kuper [17] и в публикациях вместе с ее коллегами [18]. 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.

2. Архитектура двухуровневой системы детерминированного параллельного программирования

Поскольку не существует единого набора средств, зафиксированных в языке программирования и библиотеке, обеспечивающих все случаи детерминированного параллельного программирования, входной язык должен позволять как детерминированный, так и недетерминированный код. Но чтобы прикладной программист мог ограничивать себя средствами гарантированно детерминированного программирования, входной язык системы оказывается двухуровневым, где код четко подразделяется на две части – гарантированно детерминированную и потенциально недетерминированную:

- верхний уровень – детерминированная часть: прикладной код на подмножестве языка, который пишет эксперт в данной предметной области. 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 предыдущем разделе 1.4, их состояние меняется монотонно на некоторой полурешетке, которую можно построить по коду класса.

Входным языком такой системы программирования может быть любой объектно-ориентированный язык со средствами параллельного и конкурентного программирования, у которого можно выделить функциональное подмножество. Такими 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 степень свободы предлагать пользователям разные механизмы реализации параллельных процессов, тредов, легких тредов (*англ.* light-weight thread).

Мы проводим прототипирование системы, используя платформу JVM[2], языки Java и Kotlin[3], язык собственной разработки Ajl [3], библиотеку Quasar[4], реализующую легкие потоки («файберы» и «стренды», *англ.* strand) на JVM, отслеживаем развитие проекта Loom[5], идуще-

[2] JVM = Java Virtual Machine.
[3] http://kotlin.jetbrains.org/
[4] https://github.com/puniverse/quasar
[5] https://wiki.openjdk.java.net/display/loom/Main

го на смену Quasar'у с целью более тесной интеграции с языком Java и последующего включения стандартную Java-библиотеку. Разработку ведем в IDE[6] Eclipse[7].

3. Значение идемпотентности для надежности распределенных систем

Помимо детерминированности есть еще одно понятие, которое из узко математического термина превращается во всё чаще упоминаемое ценное свойство параллельных и распределенных программ: *идемпотентность*[8]. 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 ее результат вместо неполученного результата первого вызова. 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[9] 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 и детерминированными, и идемпотентными. По нашему опыту разработки монотонных классов, сначала легче заботиться об идемпотентности, а после этого детерминированность часто получается сама собой или нужно приложить лишь немного усилий, чтобы ее достичь.

4. Понятие монотонного класса и объекта

Резюмируем определение монотонности классов и объектов, как оно используется в нашей разработке и было дано в предыдущих работах [14, 8, 6, 9]. Оно дается не через наличие полурешетки, как в [17, 18], а операционно.

[6] IDE = Integrated Development Environment.
[7] https://www.eclipse.org/
[8] https://en.wikipedia.org/wiki/Idempotence
[9] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Монотонные классы и объекты таковы, что при их создании и 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.

Обратите внимание на следующую тонкость этого определения: монотонность классов и объектов определяется не через их свойства как таковых, а через свойства *любой* программы, использующей эти классы, причем язык программирования должен быть функциональным или близким к нему по основным свойствам. 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.

5. Пример простого монотонного класса

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 (*англ.* I-structures) [10].

I-структурой называется переменная целочисленного типа (или другого примитивного, то есть нессылочного) со следующей семантикой операций создания переменной, записи и чтения из нее:

– при создании I-структура находится состояние «не определено»;
– при записи значения *x*:
  - если состояние переменной «не определено», *x* записывается в переменную;
  - если значение переменной было равно *x*, то ничего не делается;
  - в противном случае вырабатывается исключение;
– при чтении:
  - если I-структура имеет состояние «не определено», процесс, вызвавший эту операцию, переходит в состояние ожидание записи в эту переменную каким-либо процессом;
  - в противном случае выдается текущее значение переменной.

На рис. 1 изображен код на языке Java класса `I_Structure` с операциями записи `set` и чтения `get` с описанной семантикой. Класс реализован с использованием стандартных средств многопоточного программирования языка Java. Булевская переменная `defined` указывает, определено ли значение поля `value`, в которое записывается аргумент операции `set`.

Подъязыком верхнего уровня может быть функциональное подмножество языка Java, на котором разрешено использовать асинхронные вызовы функций средствами API `java.util.concurrent`. Из операций этого API допустимы лишь такие действия:

- создать асинхронный вызов с моментальным запуском на выполнение;
- ждать результата вычисления вызова методом `Future.get()`.

Как-либо опрашивать и узнавать состояние незавершенного `Future`-объекта не разрешается.

Тогда класс `I_Structure` является монотонным, то есть гарантирует детерминированность и идемпотентность любой такой функциональной программы.

```
class I_Structure {

  private boolean defined = false;
  private int value;

  public synchronized int get()
  {
    if (!defined) wait();
    return value;
  }

  public synchronized void set(int x)
  {
    if (!defined) {
      value = x;
      defined = true;
      notifyAll();
    }
    else if (value != x)
      throw new RuntimeException();
  }
}
```

Рис. 1. Монотонный класс I_Structure с одним полем примитивного типа `int` на языке Java.

6. Примеры задач, решаемых с монотонными объектами

Для изучения и демонстрации возможностей программирования с монотонными объектами в настоящее время мы обкатываем систему и накапливаем библиотеку монотонных класса на следующих классах задач.

6.2. Порождение и обработка графов

Классические чисто функциональные языки не дают возможности обрабатывать графы эффективно, когда узлы 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 столько же эффективного представления графа средствами ограниченных побочных эффектов, какие допустимы для монотонных объектов, – нетривиальная задача. Посмотрим, что будет если в классе на рис. 1 заменить тип переменной `value` с примитивного `int` на ссылочный `Object`. Класс станет немонотонным. Чтобы показать это, рассмотрим такую последовательность предложений:

```
I_Structure a = new I_Structure();
a.set(new I_Structure());          // (1) первое вычисления выражения
a.set(new I_Structure());          // (2) второе вычисление такого же выражения
```

Строки (1) и (2) содержат одинаковые выражения, и идемпотентность требует, чтобы второе выражение (2) вычислялось с таким же результатом, как выражение (1), и без нового побочного эффекта. Однако, выполнение (2) приведет к выработке `RuntimeException`, так как после выполнения (1) полю `a.value` будет присвоена ссылка на объект, созданный выражением `new I_Structure()` на строке (1), а второй вызов попытается записать в тот же

поле ссылку на новый объект `new I_Structure()`, созданный при выполнении строки (2), а она неравна первой.

В работе [9] эта проблема разбирается подробнее и предлагаются 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, претендующим на монотонность (похожем на `I_Structure` 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. Однако операция `set` позволяет 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 `RuntimeException`. Можно убедиться, что классы и объекты с такой операцией записи `set` и такой фабрикой одновременного создания объектов являются монотонными.

У такого решения есть недостаток, что 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.

6.2. 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. Оптимизационные задачи типа поиска кратчайшего пути в графе методом ветвей и границ (*англ*. branch-and-bound) принадлежат этому классу.

По смыслу задачи в них присутствуют несколько видов монотонно изменяемых данных: монотонно меняется рекорд (кратчайший путь, найденным 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. Желательно, чтобы «переключение» реализации с одного варианта на другой делалось в одном месте кода, чтобы упростить доказательство их эквивалентности. Гарантии монотонности и эквивалентности должны быть реализованы в как можно меньшем по размеру коде монотонных классов.

С точки зрения методов разработки монотонных классов, это 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.

4. Заключение

В статье дана мотивировка и принципы построения системы детерминированного параллельного объектно-ориентированного программирования, разрабатываемой авторами на основе понятия монотонных классов объектов. Представлен обзор предшествующих работ, идеи которых используются в данной разработке. Описаны два класса прикладных задач, на которых система отрабатывается и будет продемонстрирована в будущих публикациях.

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 Анд. 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 Анд. 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 Анд.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. / Ин-т систем информатики им. А. П. Ершова СО РАН, Новосиб. гос. ун-т. Новосибирск: ИПЦ НГУ, 2019. 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 Анд. 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. *Adamovich A.I.*, *Klimov And.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 / eds. V. Zakharov, N. Shilov, I. Anureev. Novosibirsk: A. P. Ershov Institute of Informatics Systems, 2019. P. 11-19. ISBN 978-5-4437-0918-5.
10. *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.
11. *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. P. 4–4 (6 pages).
12. *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.
13. *Hammond K.*, *Michelson G.* (eds.). Research Directions in Parallel Functional Programming, London, UK: Springer, 1999.
14. *Klimov And. V.* Dynamic Specialization in Extended Functional Language with Monotone Objects // SIGPLAN Not. 1991. V. 26, № 9. P. 199–210. DOI: 10.1145/115865.376287.
15. *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.

16. *Kahn G.* The Semantics of a Simple Language for Parallel Programming // IFIP Congress, 1974. P. 471–475.
17. *Kuper L.* Lattice-based Data Structures for Deterministic Parallel and Distributed Programming, Ph.D. Thesis. IN, USA: Indiana University, 2015. 253 p.
18. *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.
19. *Marlow S.* Parallel and Concurrent Programming in Haskell. CA, USA: O'Reilly, 2013.

Статья поступила в редакцию 31.07.2019;
переработанный вариант — 00.00.2019

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, г. Москва, Миусская пл., д. 4) тел. (4852) 695-228, e-mail: klimov@keldysh.ru.

An Approach to Construction of a Deterministic Parallel Programming System Based on Monotonic Objects

A. I. Adamovich, And. 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 how to make 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.