Санкт-Петербургский государственный университет

Программная инженерия
Системное программирование

Семин Алексей Васильевич

Разработка оптимизирующего компилятора языка DxEL на платформе Java

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор А. Н. Терехов

Рецензент:
Д. И. Цителов

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering
Software Engineering

Aleksei Semin

Implementation of DxEL optimising compiler for Java platform

Graduation Thesis

Scientific supervisor:
professor Andrey Terekhov

Reviewer:
Dmitry Tsitelov

Saint-Petersburg
2016

Оглавление

Введение **4**

1. Обзор **6**

1.1. Векторно-ориентированные языки 7

1.2. Обработка сложных событий 9

1.3. Предметно-ориентированные языки 9

1.4. Вывод 11

2. Постановка задачи **12**

3. Язык dxScript (DxEL) **13**

4. Реализация **15**

4.1. Архитектура библиотеки 16

4.2. Интерфейс исполняемого индикатора 18

4.3. Компилятор 19

4.4. Тестирование 25

5. Замеры производительности **26**

5.1. Производительность оптимизирующего компилятора . . 26

5.2. Сравнение с существующими решениями 27

6. Апробация **28**

6.1. Консольное приложение 28

6.2. Интеграция с InfoRider 29

Заключение **30**

Список литературы **31**

Приложение **33**

Введение

Область финансов, с момента ее возникновения, является важным объектом исследования в математике. Вместе с расширением влияния финансовой сферы активно развивались способы ее изучения. Появление все более эффективных и надежных способов коммуникации различным образом влияло на разработку методов изучения финансов. Растущие объемы передаваемой информации, с одной стороны, предоставляли больше данных для проверки экономических теорий, с другой стороны, препятствовали извлечению действительно полезных сведений.

Приход компьютерных технологий в данную область также не стал исключением. Биржи получили возможность проведения несравнимо большего количества операций по сравнению со временем, когда весь процесс полностью опирался на людей. Данное обстоятельство привело к необходимости применения компьютерных наук в финансовой аналитике, поскольку сами люди уже не были способны вручную проводить необходимые вычисления.

Одной из существующих методологий анализа финансовых данных является технический анализ [12]. Технический анализ используется для предсказания поведения некоторой численной величины на основе ее предыдущих значений. Примером таких величин могут служить цены акций на бирже. Значения величины в различные моменты времени образуют временной ряд. Временные ряды долго и активно изучаются как в математике, так и в ее приложениях [4], [15]. Для анализа временных рядов в данной методологии применяются *технические индикаторы. Технический индикатор* - это математическое вычисление, позволяющее извлечь из временного ряда скрытую информацию, которая в последствии используется техническими аналитиками для принятия решений.

Способы расчета технических индикаторов также менялись со временем. До появления компьютеров вычисления проводились вручную на миллиметровой бумаге. Когда машины научились рассчитывать ма-

тематические формулы, серьезной проблемой был ввод и вывод данных, поскольку он осуществлялся с помощью бумажных носителей. В эпоху компьютеризации финансовые данные стали не только храниться, но также появляться изначально в цифровом виде. Начиная с того времени, популярность набирали табличные процессоры, такие как Microsoft Excel [1], где элементами математических формул являлись части столбцов таблицы.

Однако в последние годы более широкое распространение получил новый подход, состоящий в применении языков программирования для описания технических индикаторов. Языки программирования позволяют отделить алгоритм вычисления индикаторов от самих данных. Также они дают возможность техническим аналитикам описывать математические преобразования на понятном им языке без необходимости иметь углубленные знания в устройстве вычислительных машин и принципах их работы.

В связи с этим встает задача реализации инструментов, с помощью которых программы на данных языках могут исполнены, чтобы предоставить специалисту из области желаемые результаты. У технического аналитика имеется ряд требований к языку как к инструменту, которым он пользуется. Помимо субъективных требований, таких как краткость и читаемость кода на данном языке, важным аспектом являются потребляемые при исполнении ресурсы. Самыми значимыми ресурсами являются время исполнения и необходимая память, поэтому при реализации подобного инструмента необходимо уделять особенное внимание данным показателям.

[1] `products.office.com/en-us/excel`

1. Обзор

К общей задаче технического анализа на данный момент применяется несколько подходов с использованием языков программирования. Технический анализ позволяет получать результаты посредством обработки временных рядов, то есть дискретных последовательностей данных или серий. Поэтому языки, с помощью которых тем или иным образом реализуются технические индикаторы, должны предоставлять средства применения заданных вычислений к сериям данных.

Необходимо явно обозначить несколько критериев, в отношении которых можно сравнивать языки применяемые в техническом анализе.

Первый критерий исходит из того, что пользователями языка не обязательно являются опытные программисты, но чаще технические аналитики. Описание технических индикаторов как чисто математических преобразований, напрямую не связано с более низкоуровневыми операциями, такими как чтение и запись данных. Поэтому пользователи не обязаны иметь углубленные знания о реализации этих операций в конкретной системе. Сами системы, в силу объемов информации и трудозатратности ее копирования, предоставляют различным пользователям доступ к одинаковым хранилищам данных. Таким образом, важной характеристикой языка является его возможность обезопасить как данные, так и управляющую систему от намеренных или случайных действий пользователя, выраженных кодом написанной им программы.

Следующие критерии вытекают из необходимости в техническом анализе оперировать над произвольными сериями. Под этим, в общем случае, подразумевается, что количество элементов серии, а также значение любого элемента может быть произвольным. Из этого, в частности, следует, что количество элементов может быть неограниченным. Возвращаясь к тому, что технические индикаторы реализуются на языках программирования, необходимо помнить, что соответствующие им программы будут исполняться на физических устройствах, количество ресурсов в которых заведомо ограниченно. Два требующих рассмотрения ресурса - это необходимая для проведения вычислений память и

время исполнения.

Таким образом, второй критерий состоит в заведомой ограниченности по памяти произвольной программы, описывающей технический индикатор. Если до запуска программы имеется возможность установить данное ограничение, это означает, что количество требуемой памяти не зависит ни от размера серии, ни от ее значений. Напротив, если память требуемая индикатором не ограниченна, то при обработке произвольного элемента серии она может закончиться на физическом устройстве, что приведет к аварийному завершению программы.

Последний критерий строится похожим образом в отношении времени, необходимого для проведения вычисления. Так как для неограниченной серии время заведомо не может быть ограниченным, то имеет смысл говорить о времени, необходимом для проведения вычисления для одного элемента серии. Если данное время ограничено, из этого следует, что любое конечное число элементов серии будет обработано за некоторое конечное время. Другими словами, пользователь гарантированно получит результат.

Отдельно необходимо отметить такую характеристику языка как привязка к торговой платформе. Многие языки, используемые в техническом анализе, разрабатываются исключительно в рамках коммерческого продукта и не могут быть использованы за его пределами.

Далее приводится краткий обзор современных языковых средств для технического анализа, результат которого приведен в таблице 1.

1.1. Векторно-ориентированные языки

Векторно-ориентированные языки представляют серии в виде векторов или массивов. Данный класс языков позволяет решать более широкий набор задач и используется не только в техническом анализе. Наиболее ярким примером здесь является язык R [2], применяемый для статистических вычислений. R завоевал значительную популярность во многом из-за существования нескольких свободно-доступных сред

[2] `www.r-project.org/about.html`

	dxScript (DxEL)	ThinkScript	EasyLanguage	MQL4/5	EPL	K	R
Безопасность	+	+	+	-	+	-	-
Ограниченность по памяти	+	+	+*	-	+	-	-
Гарантированная завершаемость	+	+	+*	-	+	-	-
Отсутствие привязки к торговой платформе	+	-	-	-	+	+	+

Таблица 1: Сравнения языков применяемых в техническом анализе

разработки и широкому набору математических библиотек. Из чего следует, отсутствие его привязки к какой-либо торговой платформе. Тем не менее R не удовлетворяет остальным рассматриваемым критериям. Пользователям языка необходимо явным образом описывать загрузку данных и сохранение результатов, так что ошибка на данных шагах, несвязанных с техническим анализом, приводит к невозможности продолжать работу. Также отличительной особенностью векторно-ориентированных языков является необходимость предварительной загрузки всех обрабатываемых данных в память вычислительной машины, что в общем случае невозможно при обработке серии произвольного размера.

Существуют другие языки для обработки массивов данных, разработанные непосредственно для технического анализа. Представителем этой группы является проприетарный язык K [3], для которого существует открытая реализация [4]. K рассчитан на использование в связке со специальной базой данных kdb+ [5]. Вследствие чего, такое решение страдает от недостатков всех векторно-ориентированных языков, в том

[3] en.wikipedia.org/wiki/K_(programming_language)

[4] github.com/kevinlawler/kona

[5] kx.com/software.php

числе, необходимости предварительной загрузки всех обрабатываемых данных.

1.2. Обработка сложных событий

Другим подходом, решающим проблему полной предварительной загрузки данных, являются системы обработки событий (CEP [7]) и используемые в них языки. Они также применяются в техническом анализе, хотя несколько проигрывают в производительности, поскольку разработаны для решения более абстрактной задачи.

В данном подходе обрабатываемая серия представляется в виде потока данных, что позволяет системам обрабатывать произвольное количество элементов. Известным представителем таких систем является Esper и используемый в нем язык EPL [6]. Esper - это проект с открытым исходным кодом, что позволяет более широкой аудитории использовать его, в том числе, в качестве подключаемой библиотеки.

1.3. Предметно-ориентированные языки

Со временем, для проведения технического анализа и решения смежных задач стали применяться специализированные языки. В большинстве случаев конкретный язык разрабатывался для применения в соответствующей торговой платформе, к которой был привязан. В связи с этим популярность того или иного языка была неразрывно связана с успешностью самой платформы.

Рассмотрим несколько современных широко-используемых языков.

MQL4/5

Языки семейства MQL и их последние версии MQL4[7] и MQL5[8] - это языки для описания торговых стратегий для торговой платфор-

[6] www.espertech.com/esper/release-5.2.0/esper-reference/html/

[7] www.mql4.com

[8] www.mql5.com

мы MetaTrader [9] 4 и 5 соответственно, вне которой их использование невозможно. Торговые стратегии или торговые роботы - это алгоритмы, которые способны взаимодействовать с биржей путем совершения таких действий как покупка или продажа акций. В своей работе торговые стратегии опираются на результаты вычисления технических индикаторов, то есть, в данном случае, технический анализ является промежуточным инструментом. Тем не менее языки, позволяющие реализовывать торговых роботов, включают в себя все необходимые средства для описания технических индикаторов.

Поэтому языки MQL позволяют решать более общую по отношению к техническому анализу задачу. Отчасти по этой причине, они не удовлетворяет рассматриваемым критериям: в них не гарантируется ограниченность памяти или завершаемость программы на одной итерации.

EasyLanguage

EasyLanguage [14] - это язык для разработки технических индикаторов для торговой платформы TradeStation [10], к которой он так же привязан. Однако в отличие от MQL-языков, EasyLanguage не предоставляет возможности реализовывать торговые стратегии и используется исключительно в целях технического анализа.

Данный язык удовлетворяет критерию безопасности, абстрагируя пользователя от операций несвязанных с математической составляющей технического индикатора. Он также гарантирует ограниченность необходимой памяти и времи на одной итерации. Однако обе гарантии проверяются только во время исполнения. В случае, если одно из условий было нарушено, система завершает работу с ошибкой. Данный подход не предоставляет возможности убедиться в том, что произвольная программа удовлетворяет критериям, до ее запуска.

[9] `www.metaquotes.net`

[10] `www.tradestation.com`

ThinkScript

Платформа ThinkOrSwim [11], в которую встроен язык ThinkScript [13], также как и MetaTrader предоставляет возможность торговли. Однако торговля не является автоматической, поэтому ThinkScript используется только для описания индикаторов.

В отличие от всех предыдущих предметно-ориентированных языков ThinkScript удовлетворяет остальным предъявленным критериям. Однако он также привязан к своей торговой платформе и имеет закрытую реализацию.

dxScript (DxEL)

Примечание: название DxEL является устаревшим названием языка dxScript.

Язык dxScript[3] - это новый язык, разработанный для описания технических индикаторов. В нем учтены недостатки существующих решений в отношении рассматриваемых критериев. Однако на данный момент не существует инструментов, с помощью которых можно было бы исполнять программы на данном языке.

1.4. Вывод

На данный момент существует множество языков программирования применяемых для описания технических индикаторов. Однако только несколько из них удовлетворяют критериям, необходимым для решения общей задачи технического анализа. Среди последних только языки EPL и dxScript могут быть использованы без привязки к какой-либо торговой платформе. При этом язык EPL изначально предназначен для решения более общей задачи обработки сложных событий, что ведет к потерям в производительности при проведении вычислений. На основе результатов обзора ставится цель данной работы.

[11] `www.thinkorswim.com`

2. Постановка задачи

Целью данной работы является разработка библиотеки для исполнения технических индикаторов, описанных на языке dxScript.

Для достижения указанной цели необходимо выполнить следующие задачи:

- Разработать интерфейс исполняемого технического индикатора.
- Реализовать библиотеку для трансляции индикаторов описанных на dxScript в исполняемую форму.
- Произвести замеры производительности индикаторов.
- Провести апробацию предложенного решения.

3. Язык dxScript (DxEL)

Примечание: название DxEL является устаревшим названием языка dxScript.

dxScript[3] - это декларативный язык для описания технических индикаторов со строгой статической типизацией. Программа на языке состоит из последовательного объявления параметров, локальных переменных и результатов (листинги 2-7).

В теле программы могут использоваться глобально-видимые *источники данных*. Имена источников обычно совпадают с именами исходных серий данных, которые обрабатывает индикатор. *Источники данных* не требуют явного объявления в теле программы, и обращение к ним синтаксически не отличается от использования локальной переменной.

Система типов языка состоит из примитивных типов и функций. Примитивные типы включают в себя вещественные числа (real) и логические значения (bool). Функциональный тип имеет вся программа целиком. Он образуется из объявленных параметров, результатов программы и их типов.

Каждый параметр имеет значение по умолчанию. Это позволяет вызывать в коде другие программы на dxScript, устанавливая их параметры выборочно.

В языке запрещена рекурсия и само-рекурсия. Это необходимо, чтобы программа гарантированно завершалась и использовала ограниченное количество памяти.

Особенностью языка является возможность получать доступ к значениям переменных на предыдущей итерации вычисления технического индикатора, которая реализуется с помощью оператора сдвига. Так, значение x или $x[0]$ - это текущие значения переменной, $x[1]$ - предыдущие, а $x[n]$ - n-нное предыдущее значение. Значение сдвига обязано быть либо численной константой, либо зависеть только от параметров индикатора, которые являются численными константами.

Технический индикатор, помимо параметров и результатов, характеризуется таким понятием как *глубина предвыборки*. Если в техниче-

ском индикаторе используются предыдущие значения какой-либо серии, то они должны быть предварительно загружены, чтобы результат вычисления был определен. *Глубина предвыборки* - это количество необходимых элементов исходной серии для вычисления на текущей итерации.

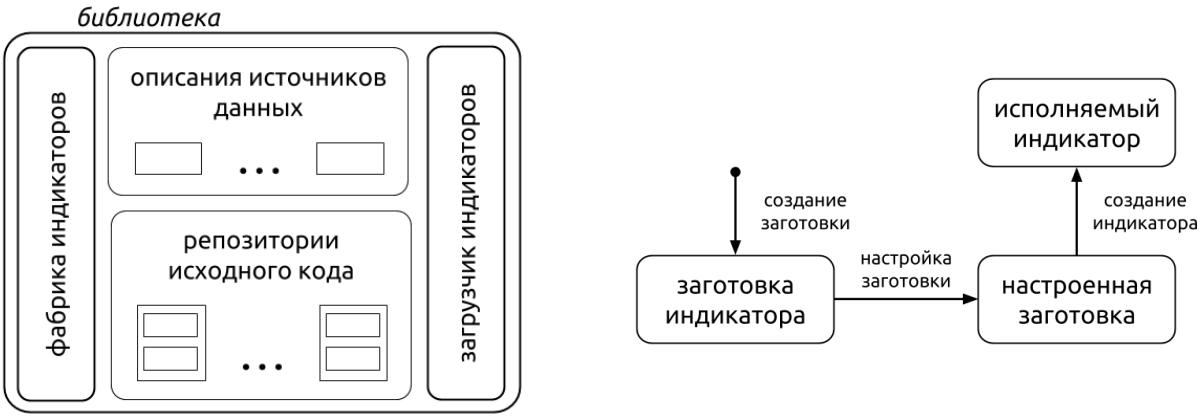
4. Реализация

Исполнение технического индикатора означает исполнение программы написанной на языке dxScript. Существует два основных подхода к исполнению программ [1]: непосредственная интерпретация или компиляция с последующим исполнением скомпилированного кода. Каждый подход имеет свои преимущества и недостатки.

В контексте данной задачи речь идет о динамическом создании исполняемого индикатора. Обычно в таких случаях применяется интерпретация, поскольку процесс предварительной компиляции слишком трудозатратный. Интерпретация позволяет практически сразу приступить к исполнению программы, в то время как компилятору необходимо сначала транслировать весь исходный код программы в код для более низкоуровневого вычислителя. С другой стороны, скомпилированный код обладает более высокой производительностью по сравнению с интерпретируемым.

Решающим фактором в пользу компиляции для решения поставленной задачи является модель вычислений подразумеваемая под техническим индикатором. Индикаторы рассчитаны на то, что будут обрабатывать потенциально неограниченные серии данных, вследствие чего, описывающие их программы будут исполнены во множественном числе итераций. Таким образом, время, необходимое на предварительную подготовку программы к исполнению, то есть время требуемое на компиляцию, в общем случае пренебрежимо мало по сравнению со временем исполнения. Более того, если дополнительное время компиляции займут какие-либо оптимизации примененные к коду, это также уменьшит время затраченное на обработку серии в целом, так как выигрыш по времени будет пропорционален количеству ее элементов.

Для реализации библиотеки и компилятора был выбран язык Java. Программы на Java исполняются виртуальной машиной JVM [11] и в скомпилированном виде хранятся как JVM-байткод. Такой подход делает код на Java и скомпилированный байткод переносимым и независимым от платформы, на которой он в конечном итоге будет исполняться.



(a) Организация библиотеки.

(b) Процесс создания индикатора.

Рис. 1: Библиотека для создания исполняемых индикаторов.

JVM имеет встроенный интерпретатор и JIT-компилятор [2]. Изначально байткод начинает исполняться интерпретатором. В течение некоторого времени JVM собирает информацию об исполнении программы, которую впоследствии может использовать, чтобы скомпилировать наиболее часто используемые методы. Из-за того, что JIT-компилятор компилирует методы во время исполнения, он может производить некоторые оптимизации кода [10], [16], которые невозможны для AOT-компиляторов [6]. В частности, это позволяет рассчитывать на подобные оптимизации в отношении кода технических индикаторов, который транслируется в JVM-байткод компилятором языка dxScript. Также JVM предоставляет встроенный функционал для динамической загрузки классов, что является удобным средством динамического создания исполняемых индикаторов.

4.1. Архитектура библиотеки

Точкой входа в библиотеку (рис. 1a) является *фабрика индикаторов*. *Фабрика* позволяет создавать исполняемые индикаторы, запрошенные по их имени. Перед использованием *фабрику* необходимо настроить.

Этап настройки *фабрики* позволяет абстрагировать ее реализацию от контекста конкретной задачи, что повышает гибкость архитектуры и позволяет использовать библиотеку в более широком классе задач.

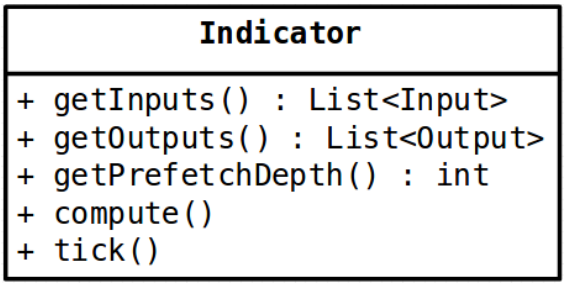
Настройка *фабрики* подразумевает задание окружения, на основе которого будут создаваться индикаторы.

В первую очередь необходимо объявить *источники данных*, которые доступны в контексте задачи, поскольку эти источники могут использоваться в исходом коде индикаторов без явного объявления в виде глобально-доступных величин.

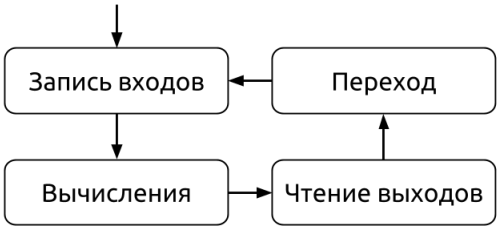
Далее *фабрике* необходимо предоставить *репозитории* исходного кода. *Репозиторием*, например, может быть директория в файловой системе. В последствии, при запросе на создание исполняемого индикатора, *фабрика* будет производить поиск его исходного кода среди представленных ей *репозиториев*.

Результатом запроса на создание индикатора является так называемая *заготовка индикатора*. С нее начинается процесс создания индикатора (рис. 1b). *Заготовка* позволяет настроить параметры индикатора, которые были описаны в его исходном коде. Часть этих параметров отвечает за *глубину предвыборки*, часть - за *источники данных*. Этап настройки заготовки имеет особенную важность в контексте приложений с графическим интерфейсом, в которых потенциально будет использоваться библиотека. В этом случае пользователю предоставляется удобный способ точно настроить индикатор перед его запуском.

Когда заготовка полностью настроена, ее параметры фиксируются и завершается процесс компиляции индикатора. Результатом компиляции является байткод Java-класса, который реализует интерфейс исполняемого индикатора. Посредством загрузчика классов Java класс индикатора динамически загружается в исполняемую JVM, позволяя создавать объекты этого класса. В результате создания объекта получается исполняемый индикатор, который содержит необходимый код для вычислений, а также выделенную память, используемую для хранения промежуточных значений. Далее исполняемый индикатор используется для выполнения последовательных вычислений над значениями исходной серии.



(a) Интерфейс индикатора.



(b) Жизненный цикл индикатора.

Рис. 2: Исполняемый индикатор.

4.2. Интерфейс исполняемого индикатора

Предложенный интерфейс индикатора представлен на рисунке 2a. Индикатор обязан содержать в себе информацию и соответствующие методы необходимые для проведения вычислений, которые неразрывно связаны с его жизненным циклом.

Жизненный цикл индикатора состоит из следующих шагов (рис. 2b): установка пользователем текущих значений источников данных, произведение индикатором вычисления, чтение пользователем результатов вычислений, изменение внутреннего состояния индикатора для перехода к следующему циклу.

Для того, чтобы получить возможность устанавливать значения *источников данных* или *входов*, пользователь вызывает соответствующий метод getInputs, который возвращает список необходимых индикатору *источников*. Аналогично для чтения результатов (*выходов*) необходимо воспользоваться методом getOutputs, который возвращает список объектов, каждый из которых соответствует отдельному результату вычисления. Подразумевается, что списки *источников* и результатов запрашиваются пользователем до начала исполнения. К тем и к другим применяется соответствующая логика привязки действий на запись и чтение данных. После чего пользователь пользуется для ввода-вывода объектами, полученными из списков, и больше не вызывает методы getInputs и getOutputs. Таким образом производятся внешние для индикатора операции.

Два оставшихся шага жизненного цикла соответствуют непосредственному проведению вычислений и переходу к следующему циклу. Вычисления производятся при помощи метода compute. При этом индикатор считывает текущие входные значения и, по окончании расчетов, делает доступными текущие результаты. После их прочтения пользователю необходимо вызвать метод tick, который отвечает за корректный переход к следующему жизненному циклу. После вызова данного метода все текущие значения становятся предыдущими, а предыдущие - пред-предыдущими, и так далее, насколько это требуется логикой программы, заложенной в индикаторе.

Также индикатор предоставляет возможность узнать точную *глубину предвыборки* с помощью метода getPrefetchDepth, который возвращает ее значение. *Глубина предвыборки* используется, чтобы пропустить несколько первых циклов, поскольку результаты вычислений не определены. Однако проведение всех остальных шагов цикла (установка входов, вычисление и переход) обязаны быть выполнены.

4.3. Компилятор

Исходя из предложенной архитектуры библиотеки, процесс компиляции индикатора делится на две части. Первая часть включает в себя предварительную обработку исходного кода с проведением лексического, синтаксического и семантического анализа. Вторая часть включает статический анализ, оптимизации и кодогенерацию.

Результатом компиляции является байткод Java-класса, который реализует интерфейс исполняемого индикатора. Все поля и методы класса образуют среду времени исполнения компилируемого индикатора.

Для доступа к предыдущим значениям серии используются кольцевые буферы. Размер буфера определяется для каждой серии во время компиляции и не меняется при исполнении. Буферы реализованы с помощью примитивных массивов Java, что позволяет избежать частой сборки мусора во время многократных итераций вычисления индикатора. Буферы представлены как поля класса, чтобы объект исполняемого

индикатора имел к ним достум из метода, производящего расчеты.

Построение заготовки

Лексический и синтаксический анализ dxScript-программ реализованы с помощью библиотеки парсер-генераторов ANTLR4[12]. Данная библиотека позволяет по грамматике языка получить анализатор, который по исходному коду программы на языке dxScript строит его абстрактное синтаксическое дерево (АСД).

На этапе семантического анализа проверяется, что все переменные объявлены до их использования, а также строится таблица символов. Инициализация переменной, в случае dxScript, происходит сразу в момент объявления. Поэтому нахождение переменной в зоне видимости означает, что она была инициализирована. Далее проводится вывод и проверка типов всех выражений. Переменные в dxScript инициализируются в момент объявления, а все параметры имеют значения по умолчанию. Поэтому становится возможным реализовать однопроходный вывод типов, опирающийся на инициализирующие выражения.

После этого имеется вся необходимая информация для построения заготовки индикатора. В процессе настройки заготовки у пользователя есть возможность менять значения параметров по умолчанию. Когда настроенные параметры зафиксированы, наступает второй этап компиляции.

Промежуточное представление

Частью заготовки индикатора является его промежуточное представление (ПП). ПП состоит из графа зависимостей (рисунок 3) и контекста. Граф зависимостей строится по АСД с использованием таблицы символов. В графе имеются узлы, которые ни от чего не зависят - это узлы констант, *источников данных* и параметров программы. Также имеются узлы, от которых не зависят другие, они соответствуют результатам программы. Сам граф содержит только информацию о

[12] `www.antlr.org`

зависимости узлов между собой, вся остальная информация об узлах хранится в контексте.

Выбор такого ПП обуславливается несколькими причинами. Поскольку в языке запрещена рекурсия и взаимная рекурсия, в графе зависимостей будут отсутствовать циклы. Это обстоятельство позволяет отсортировать вершины графа в топологическом порядке. Наличие топологического порядка дает возможность проводить необходимый статический анализ за один обход графа зависимостей. Также, обратный топологический порядок соответствует графу потока данных программы, который используется в методах статического анализа [9] и оптимизациях [5]. В данном случае, граф потока данных так же используется для кодогенерации.

Наличие контекста в ПП позволяет сильно упростить добавление статического анализа кода и применение оптимизаций. Так, на любой стадии компиляции в контекст можно добавить дополнительную информацию об узлах графа зависимостей. Это не затронет предыдущие или последующие этапы компиляции, однако данная информация может использоваться непосредственно при генерации кода, чтобы увеличить его производительность.

После фиксации параметров индикатора узлы соответствующие этим параметрам в графе зависимостей заменяются на узлы конкретных значений. Таким образом для самого индикатора и всех программ, которые он использует для вычислений, становятся явно известны все внешние зависимости. Это позволяет проводить анализ и применять оптимизации не только локально, в рамках самого индикатора, но так же межпроцедурно. Данное обстоятельство значительно сказывается на результирующем эффекте оптимизаций.

Расчет глубины предвыборки

В начале второй части компиляции необходимо провести статический анализ индикатора, с зафиксированными параметрами. В частности, анализ включает в себя проверку того, что обращения к предыдущим элементам серии происходят на конечную глубину, а также расчет

общей *глубины предвыборки* индикатора.

Предварительно в графе зависимостей производится оптимизация распространения констант (рис. 4а). Это позволяет проверить, что обращения к предыдущим значениям серии происходят на фиксированную глубину, а также рассчитать минимальный размер буфера, для хранения этих значений.

```
script example;
in  x = data;
in  n = 1;

def y = x[n-1];
def t = y * 3;

out z = y[2];
```

Листинг 1: Пример программы.

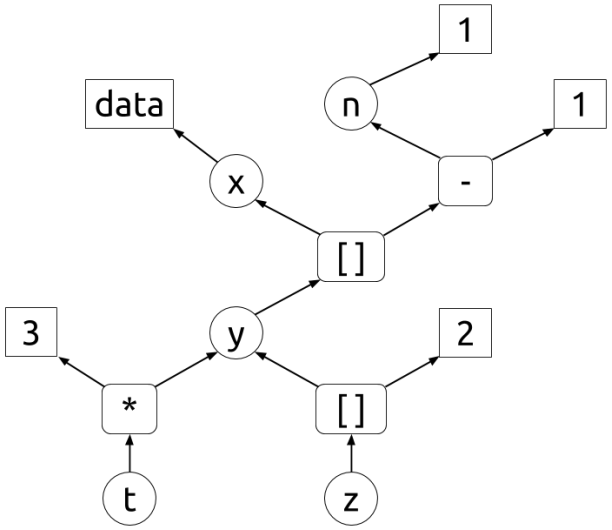
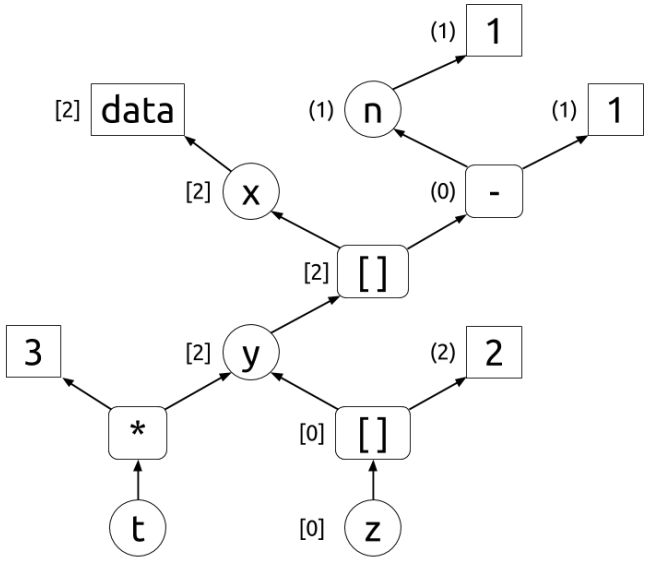


Рис. 3: Граф зависимостей программы из листинга 1.

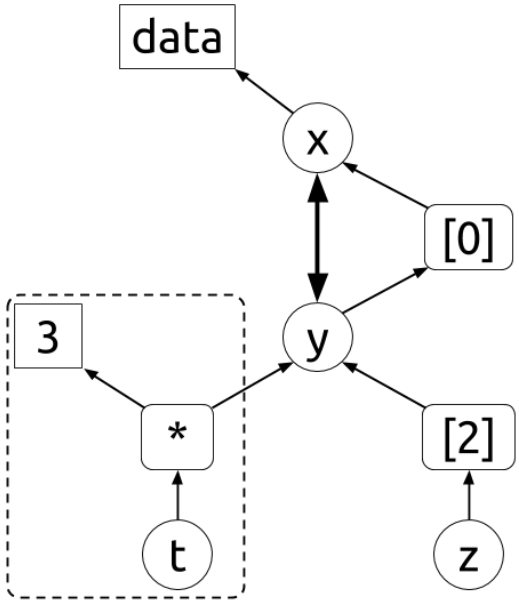
Расчет общей *глубины предвыборки* также происходит за один обход графа зависимостей в обратном топологическом порядке. При завершении обработки вершины все вершины зависящие от нее обязаны быть посещены, следовательно *глубина предвыборки* для данной вершины уже не изменится при продолжении обхода. Тогда для каждой вершины, достаточно обновить значения глубины в зависимых вершинах. Значения общей *глубины предвыборки* рассчитывается как наибольшее из значений в вершинах *источников данных* (рис. 4а).

Оптимизации

С целью ускорения работы исполняемых индикаторов для компилятора языка dxScript было реализовано несколько оптимизаций: *анализ совмещений* (alias analysis), *удаление мертвого кода* (dead code elimination) и *скаляризация* (scalarization). Пример применения опти-



(a) Распространение констант (n) и расчет глубины предвыборки $[n]$.



(b) Пунктиром обозначен скаляризованный участок, двойной стрелкой - совмещения переменных.

Рис. 4: Применение статического анализа и оптимизаций к программе из листинга 1.

мизаций показан на рисунке 4.

Удаление мертвого кода позволяет исключить из результирующего кода неиспользуемые вычисления. Такие вычисления могут быть в коде изначально, вследствие не оптимально написанного кода, а также появиться в процессе распространения констант. *Удаление мертвого кода* увеличивает производительность программы.

Анализ совмещений применяется, если в языке имеется прямой доступ к памяти, то есть указатели на память [8]. Тогда для некоторых участков программы можно утверждать, что различные указатели указывают на одну ячейку памяти. В этом случае указатели можно слить в один, что зачастую приводит к уменьшению инструкций кода в результирующей программе и дает возможность применения других оптимизаций.

В *анализе совмещений* для dxScript под указателями подразумеваются переменные указывающие на одну и ту же серию данных. По-

скольку для каждой серии во время кодогенерации необходимо создать промежуточный буфер, *анализ совмещений* позволяет уменьшить количество таких буферов. В результате это приводит к уменьшению памяти требуемой программой для исполнения.

В dxScript любая переменная может использоваться как серия данных, то есть пользователь может обратиться к ее предыдущим значениям. Тем не менее не для всех переменных это необходимо, поскольку некоторые из них используются только для хранения промежуточных результатов. Под *скаляризацией* понимается анализ использования переменных и выявление тех из них, для которых нет необходимость заводить отдельный буфер. Значения такой переменной могут храниться в локальной переменной внутри метода соответствующего вычислению индикатора. Применения такой оптимизации позволяет уменьшить количество буферов, а также ускорить программу, так как обращение к локальной переменной происходит быстрее, чем запрос значений из буфера.

Кодогенерация

Когда вся необходимая для кодогенерации информация собрана, необходимо преобразовать промежуточное представление программы в Java-байткод. Для решения данной задачи применялась библиотека ASM[13], которая предоставляет удобную функциональность генерации байткода.

Для того, чтобы уменьшить время затрачиваемое при генерации кода индикатора, часть его функциональности, общей для всех индикаторов, была реализована в родительском абстрактном классе. Остальная часть, которая индивидуальная для каждого индикатора: объявление полей класса (соответствующих буферам для хранения промежуточных результатов), инициализация полей (с учетом их точного размера, полученного из статического анализа), а также объявление метода, содержащего код вычисления.

[13] `asm.ow2.org`

4.4. Тестирование

Для проверки корректности реализации на каждом этапе разработки проводилось регрессионное тестирование. Для каждого выражения языка был реализован набор тестов, каждый из которых содержал код, использующий данное выражение. Результаты вычислений сравнивались с ручными расчетами. Также были реализованы комплексные тесты содержащие различные конструкции языка, для проверки их корректной совместной работы.

В процессе тестирования была разработана небольшая библиотека, позволяющая автоматизировать проверку корректности вычислений. Она предоставляет возможность по исходному коду, исходным данным и ожидаемым результатам полностью выполнять эту проверку. Также она проверяет корректность вывода типов, сравнивая тип индикатора с ожидаемым. Ожидаемая *глубина предвыборки* тоже входит в состав любого теста и может быть сверена с рассчитанной во время компиляции.

Такая библиотека помогает процессу разработки в целом, поскольку позволяет быстро добавлять новые тесты, а также отлаживать ошибки в случае некорректной реализации частей компилятора.

5. Замеры производительности

Для измерения разницы в производительность исходного и оптимизирующего компилятора, а также для сравнения с существующими решениями, на языке dxScript были реализованы несколько широкоиспользуемых технических индикаторов. В таблицах с результатами замеров рядом с именем каждого индикатора указаны его параметры. Данные значения параметров выбранных индикаторов являются общепринятыми в техническом анализе. Важность параметров для измерения производительности состоит в их связи с глубиной предвыборки и количестве вычислений на одной итерации

Все замеры производились на вычислительной машине с процессором Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz и оперативной памятью размером 16GB.

5.1. Производительность оптимизирующего компилятора

Для выявления улучшения производительности исполняемых индикаторов после применения оптимизирующего компилятора производились замеры среднего времени, которое необходимо индикатору для вычисления результата одной итерации. Входные данные для итерации генерировались случайным образом.

Для реализации замеров высокой точности использовалась библиотека JMH[14]. Особенность данной библиотеке состоит в том, что она позволяет производить замеры с точностью до наносекунд. Также она учитывает внутреннее устройства виртуальной машины Java (JVM), при измерении времени. В частности, это выражается в необходимости "разогреть" (warm-up) JVM, прежде чем приступать к измерениям. Наличие этого этапа позволяет избавиться от ошибок измерения связанных с работой JIT-компилятора Java.

В таблице 2 представлены результаты замеров среднего времени ис-

[14] `openjdk.java.net/projects/code-tools/jmh/`

Индикатор	Среднее время итерации (нс)		Ускорение
	Исходная версия	Оптимиз. версия	
Empty	17.386 ± 0.026	17.406 ± 0.053	1x
SMA(10)	75.177 ± 0.439	55.403 ± 0.220	1.3x
EMA(10)	69.693 ± 0.358	48.880 ± 0.563	1.4x
MACD(12,26,9)	255.366 ± 1.450	178.386 ± 0.740	1.4x
CCI(14)	368,923 ± 2.171	245.949 ± 2.845	1.5x
Stochastic(10,10,3)	543.533 ± 8.593	279.014 ± 0.943	1.9x

Таблица 2: Сравнения исполняемых индикаторов без и с применением оптимизаций при компиляции

Индикатор	Расчет 100000 итераций (мс)		
	dxScript	dxScript (оптим.)	ThinkScript
MACD(12,26,9)	26.205 ± 3.314	21.788 ± 3.457	116.028 ± 0.740
CCI(14)	30.572 ± 2.742	23.892 ± 1.407	212.62 ± 10.684
Stochastic(10,10,3)	31.61 ± 3.998	25.19 ± 2.116	151.605 ± 4.117

Таблица 3: Сравнения исполняемых индикаторов с существующими решениями

полнения одной итерации для индикаторов скомпилированных без и с оптимизациями, а также рассчитано ускорение. Для каждого замера было произведено 3 ”разогрева” по 1 секунде и 10 замеров также по 1 секунде. В течение одного замера, индикатор непрерывно обрабатывал входные данные. Таким образом при одном замере индикатор совершал множество итераций. Результатом замера является среднее время, потраченное исполняемым индикатором на итерацию.

5.2. Сравнение с существующими решениями

Также были произведены замеры времени необходимого для обработки 100000 итераций, на данных загруженных в память. На основе данных замеров было произведено сравнение с производительностью таких же индикаторов, реализованных на языке ThinkScript.

Для измерения времени производилось 100 запусков для каждого замера, из которых рассчитывалось среднее время. Результаты замеров приведены в таблице 3.

6. Апробация

В целях апробации предложенного решения было реализовано консольное приложение, а также произведена интеграция с продуктом InfoRider.

6.1. Консольное приложение

В качестве первой апробации разработанной библиотеки, а также для дальнейшего использования в разработке, было реализовано консольное приложение. Данное приложение представляет собой интерфейс командной строки (command-line interface) для библиотеки. Основной задачей приложения является расчет технических индикаторов.

Для исполнения индикатора приложению необходимо описание самого индикатора, а также данные, на которых индикатор будет рассчитан. Пользователь может указать имя индикатора при запуске приложения. Далее будет произведен поиск файла с исходным кодом данного индикатора в текущей или вложенных директориях.

Данные для расчета индикатора приложение получает из входного файла, путь к которому также необходимо указать при запуске. Необходимо, чтобы файл с данным был в формате CSV (comma-separated values) и в первой строчке содержал названия колонок. Каждая из колонок при исполнении считается источником данных для индикатора. По умолчанию результат вычисления выводится прямо в консоль, однако при необходимости вывод можно перенаправить в файл, воспользовавшись опцией приложения. Также, приложение выведет справку с перечислением всех доступных опций, если запустить его без аргументов.

Во время исполнения строчки входного файла последовательно считываются, и значения, находящиеся в них, подаются на вход исполняемому индикатору. После этого происходит печать текущего результата. Стоит отметить, что при данной модели для приложения не имеет значения размер входного файла, поскольку он читается последовательно один раз.

6.2. Интеграция с InfoRider

InfoRider[15] - это продукт компании Devexperts, для торговли и аналитики биржевых данных. Программный продукт представляет собой графическое приложение. Одной из основных задач такого инструмента является наглядное отображение данных и их аналитики, в том числе результаты вычисления технических индикаторов.

В продукте построен следующий процесс расчета индикаторов и отображения результатов их вычисления в графическом интерфейсе. Пользователь может выбрать набор индикаторов, которые он хотел бы отобразить на графике. Данные для расчета индикаторов поставляются биржей или другими источниками посредством отдельных отвечающих за это частей приложения. Каждый индикатор оповещается о поступлении новых данных, после чего внутри индикатора происходит расчет результатов. В то же время, компоненты графического интерфейса следят за появлением новых результатов вычислений, отображая их на экране.

Архитектура библиотеки, реализованной в данной работе, а также предложенный интерфейс исполняемых индикаторов естественным образом интегрируются в описанный процесс. Наборам индикаторов, из которых пользователь может выбирать отображаемые, соответствуют репозитории, содержащие исходные описания индикаторов. Все выбранные для отображения индикаторы транслируются библиотекой в исполняемую форму с помощью компилятора. Далее каждый индикатор исполняет свой жизненный цикл. При появлении новых данных, которые необходимо обработать, они устанавливаются в текущие значения входов индикатора. После чего, происходит расчет, чтение выходов и переход к следующей итерации. Результат вычислений становится доступен графическому интерфейсу для отображения, а переход обеспечивает корректность дальнейшей работы индикатора.

[15] `www.devexperts.com/en/market-data/inforider.html`

Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Разработан интерфейс исполняемого технического индикатора.
2. Реализован оптимизирующий компилятор языка dxScript на платформе Java.
3. Реализована библиотека трансляции индикаторов на языке dxScript в исполняемую форму.
4. Проведена апробация решения в виде реализации консольного приложения для расчета технических индикаторов и интеграции с продуктом InfoRider компании Devexperts.
5. Произведены замеры эффективности исполняемых индикаторов, показавшие средний прирост производительности в 30-90% при использовании оптимизирующего компилятора, а также превосходство над существующими решениями.
6. Промежуточные результаты работы были представлены в докладе на конференции "Программная Инженерия и Организация Информации"[16].

[16] `seim-conf.org`

Список литературы

[1] Aho Alfred V, Sethi Ravi, Ullman Jeffrey D. Compilers, Principles, Techniques. — Addison wesley, 1986.

[2] Compiling java just in time / Timothy Cramer, Richard Friedman, Terrence Miller et al. // Micro, IEEE. — 1997. — Vol. 17, no. 3. — P. 36–43.

[3] Devexperts. dxScript Specification. — Devexperts, 2016. — URL: `docs.dxfeed.com/dxscript/dxScriptSpecification.pdf` (online; accessed: 25.05.2016).

[4] Hamilton James D. Time Series Analysis. — Princeton University Press, 1994.

[5] Jones Neil D, Muchnick Steven S. Flow analysis and optimization of LISP-like structures // Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages / ACM. — 1979. — P. 244–256.

[6] Practical experiences with Java compilation / Todd Smith, Suresh Srinivas, Philipp Tomsich, Jinpyo Park // High Performance Computing—HiPC 2000. — Springer, 2000. — P. 149–157.

[7] Robins D. Complex event processing // Second International Workshop on Education Technology and Computer Science. Wuhan. — 2010.

[8] Ruf Erik. Context-insensitive alias analysis reconsidered // ACM SIGPLAN Notices. — 1995. — Vol. 30, no. 6. — P. 13–22.

[9] Sharir Micha, Pnueli Amir. Two approaches to interprocedural data flow analysis. — New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[10] Su Lixin, Lipasti Mikko H. Speculative optimization using hardware-monitored guarded regions for java virtual machines // Proceedings of

the 3rd international conference on Virtual execution environments / ACM. — 2007. — P. 22–32.

[11] T. Lindholm F. Yellin G. Bracha A. Buckley. The Java Virtual Machine Specification. — Oracle America, Inc., 2015. — URL: docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (online; accessed: 25.05.2016).

[12] Taylor Mark P, Allen Helen. The use of technical analysis in the foreign exchange market // Journal of international Money and Finance. — 1992. — Vol. 11, no. 3. — P. 304–314.

[13] ThinkOrSwim. ThinkScript Learning Center. — 2016. — URL: goo.gl/RfJuPT (online; accessed: 25.05.2016).

[14] TradeStation Securities Inc. EasyLanguage Essentials. — TradeStation Securities, Inc., 2007. — URL: goo.gl/FSzFk2 (online; accessed: 25.05.2016).

[15] Wei William W.S. Time Series Analysis : Univariate and Multivariate Methods. — Pearson, 2005.

[16] A general compiler framework for speculative optimizations using data speculative code motion / Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew // Proceedings of the international symposium on Code generation and optimization / IEEE Computer Society. — 2005. — P. 280–290.

Приложение

В данном приложении представлен исходный код функций и технических индикаторов на dxScript.

```
script change;
in  x = 0;
out y = x - x[1];
```

Листинг 2: Функция change.

```
script abs;
in a = 0;
out b = if (a > 0) a else -a;
```

Листинг 3: Функция abs.

```
script lindev;
in series = 0;
in len = 12;
def avg = sma(series, len);
def d = abs(avg - series);
out v = sma(d, len);
```

Листинг 4: Функция lindev.

```
script CCI;
in len = 14;
in over_sold = -100;
in over_bought = 100;

def price = close+low+high;
def ld = lindev(price, len);
def d = price - sma(price, len);

out Value = if (ld == 0) 0
            else d/ld/.015;
out OverBought = over_bought;
out ZeroLine = 0;
out OverSold = over_sold;
```

Листинг 5: Индикатор CCI.

```
script MACD;
in fast = 12;
in slow = 26;
in n = 9;

// `close` is a data-source
def x = close;
out Value = ema(x, fast)
          - ema(x, slow);

out Avg = ema(Value, n);
out Diff = Value - Avg;
out ZeroLine = 0;
```

Листинг 6: Индикатор MACD.

```
script StochasticFull;
in over_bought = 80.0;
in over_sold = 20.0;
in K = 10;
in D = 10;
in S = 3;

def lk = lowest(low, K);
def c1 = close - lk;
def c2 = highest(high, K)-lk;
def fastK = if (c2 == 0) 0
            else c1/c2*100;

out FullK = sma(fastK, S);
out FullD = sma(FullK, D);

out OverBought = over_bought;
out OverSold = over_sold;
```

Листинг 7: Индикатор Stochastic.