ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №4(31), 2016, с. 83–117

УДК 519.685.1+519.682.2

А. И. Адамович

Язык программирования Ajl: автоматическое динамическое распараллеливание для платформы JVM

Аннотация. Нынешнее состояние программного обеспечения и аппаратных средств широкого применения настойчиво требует развития инструментов параллельного программирования на основе языка Java. В ИПС им. А.К. Айламазяна РАН выполнена реализация языка Ajl, являющегося расширением языка Java и предназначенного для разработки параллельных программ на основе использования вычислительной модели «самотрансформация вычисляемой сети». В данной публикации рассматриваются различные аспекты выполненной работы — от синтаксиса и семантики конструкций языка и методов разработки транслятора до реализации базовых понятий использованной модели вычислений. Описывается экспериментальное исследование выполненной реализации и обсуждаются его результаты. В заключение приводится краткий обзор близких работ.

Ключевые слова и фразы: реализация языков программирования, параллельные вычисления, язык Java, автоматическое динамическое распараллеливание, струи.

Введение

Мысль о необходимости создания и развития инструментальных средств для высокопродуктивной разработки параллельных программ стала за последнее десятилетие общим местом. С появлением таких широкодоступных параллельных аппаратных платформ, как многоядерные планшеты и смартфоны, эта идея обрела новое измерение.

Исследование выполнено в рамках НИР «Методы и средства разработки эффективного программного обеспечения, ориентированные на вычислительные системы, построенные на основе микропроцессоров с многоядерной архитектурой, и формальные основы высокоуровневых языков программирования для суперкомпьютеров с гибридной архитектурой» (№ г/р 01201455360) (госзадание ФАНО России).

Потребность в распараллеливании программ для мобильных платформ привела к необходимости отказа от устойчивого некогда в области параллельных вычислений принципа “хочешь написать параллельную программу — пиши ее на C (Фортране, Ассемблере)”. Основным языком разработки программ для наиболее распространенной мобильной ОС Android является язык Java, и Java-разработчики не желают терять в продуктивности, переходя на языки программирования, заметно отличающиеся от Java[1]. Похожая ситуация возникла и в области распределенных программных архитектур, где серверная часть приложения зачастую реализуется на Java, что, по мнению разработчиков, обеспечивает повышенный уровень надежности.

В то же время, возможности реализации параллельных программ, предоставляемые библиотекой языка Java, имеют недостатки, характерные для большинства средств ручного распараллеливания: достаточно высокую трудоемкость разработки и подверженность ошибкам. Попыткой найти решение данной проблемы является разработка в ИПС им. А. К. Айламазяна РАН экспериментального языка программирования Ajl.

Этот язык, с одной стороны, основан на использовании парадигмы «автоматическое динамическое распараллеливание программ», тем самым облегчая разработку параллельных прикладных программ и повышая качество распараллеливания. Реализация в языке Ajl указанной парадигмы распараллеливания основывается на подходе, использующем оригинальную модель вычислений «самотрансформация вычисляемой сети» [**1**, **2**]; с момента своего появления данный подход прошел значительный путь развития (см., например, [**3**–**5**]).

С другой стороны, язык Ajl построен как расширение языка Java небольшим набором дополнительных конструкций, тем самым значительно облегчается его использование программистами, привыкшими к использованию языка Java.

Данная публикация посвящена рассказу о результатах данной — успешной, как нам кажется — попытки: о свойствах полученного языка и найденных методах его реализации.

Дальнейшее изложение будет следовать следующему плану: в первом разделе описывается синтаксис и семантика конструкций

[1]Впрочем, широкое распространение многоядерных процессоров привело к тому, что средства организации параллельного исполнения программ проникли и во многие другие языки программирования, нехарактерные для области параллельных вычислений.

языка Ajl, расширяющих язык Java. Затем во втором разделе излагаются принципы реализации основных понятий вычислительной модели «самотрансформация вычисляемой сети». В следующем, третьем разделе приводится описание эксперимента, направленного на изучение свойств выполненной разработки, а также обсуждаются его результаты. И, наконец, в четвертом разделе приводится небольшой обзор близких работ.

1. Конструкции языка Ajl: синтаксис и семантика

Языковые конструкции Ajl отражают понятия модели вычислений «самотрансформация вычисляемой сети», поэтому логично было бы напомнить читателю ее основные положения. Именно этому и посвящен следующий подраздел.

1.1. Модель вычислений «самотрансформация вычисляемой сети»

Модель организации вычислений «самотрансформация вычисляемой сети» основывается на следующих базовых принципах:

- В качестве основной парадигмы рассматривается функциональное программирование. Программа представляет собой набор чистых, без побочных эффектов, функций (*T-функций*). Каждая T-функция может принимать несколько аргументов и возвращать несколько результатов. В то же время тела T-функций могут быть описаны в императивном стиле (на языках типа FORTRAN, C и т. п.) и могут включать в себя, в том числе, вызовы обычных, регулярных, функций входного языка. Важно только, чтобы:
  - всю информацию извне T-функция получала только через свои аргументы;
  - вся информация из T-функции передавалась только через явно описанные результаты.
- Рассмотрим вызов некоторой T-функции `G` (принимающей два аргумента и возвращающей три результата):

T-System

```
1  [x,y,z] = G(a,b)
```

Данный вызов, производимый в процессе вычисления другой T-функции `F`, выполняется нетрадиционным способом (так называемый сетевой вызов функции). При этом порождается новый T-процесс с несколькими входами (их число соответствует числу

аргументов Т-функции `G`) и несколькими выходами (их число соответствует числу результатов Т-функции `G`). Выходы нового Т-процесса связываются с соответствующими переменными Т-процесса `F` (Т-переменными или, иначе выражаясь, *внешними* переменными) — в нашем примере это Т-переменные `x`,`y`,`z` — отношением «поставщик-потребитель», и тем самым Т-переменные-потребители принимают неготовые (то есть ещё не вычисленные) значения. Порожденный Т-процесс `G` должен вычислить Т-функцию `G` и заменить неготовые значения у всех своих потребителей на соответствующие результаты Т-функции `G`.

- Все Т-процессы, вычисляющие Т-функции, порождаются в состоянии «готов к вычислению». Так, например, описанный выше вызов Т-функции `G` из Т-процесса `F` не ведет к потере Т-процессом `F` *готовности*, т.е. способности выполняться. Единственное событие, приводящее к потере готовности (к засыпанию) Т-процесса — это попытка Т-процесса выполнить с неготовым значением любую операцию, отличающуюся от операций передачи значений; операции передачи неготовых значений не приводят к потере процессом готовности и реализуются за счет изменения вычисляемой сети.
- Если некоторый Т-процесс предпринял попытку выполнить с неготовым значением операцию, отличную от операции передачи данных, то такой Т-процесс теряет готовность — засыпает (приостанавливается). Готовность Т-процесса будет восстановлена — Т-процесс будет разбужен — в тот момент, когда соответствующее неготовое значение (причина засыпания) будет заменено поставщиком на обычное значение.
- Таким образом, в каждый момент времени состояние вычисления представлено вычисляемой сетью, узлами которой являются запущенные Т-процессы и структуры данных, а дугами — отношения поставщик-потребитель. в процессе выполнения программы вычисляемая сеть самотрансформируется: в моменты вызовов Т-функций в ней появляются новые узлы, в моменты завершения вычисления всех потребляемых результатов Т-функций узлы исчезают, при выполнении допустимых операций с неготовыми значениями появляются и исчезают дуги.

Наличие в вычисляемой сети достаточного количества готовых к выполнению Т-процессов позволяет выполнять их параллельно.

1.2. Подход к построению языка Ajl

Язык Ajl построен как «расширенное сужение» языка Java. С одной стороны («расширение»), в базовый язык добавляются новые конструкции, которые отражают понятия используемой вычислительной модели, такие, как T-переменная, T-функция, вызов T-функции и рассылка (возврат) одного из ее результатов.

С другой стороны («сужение»), в Ajl обеднены — по сравнению с Java — возможности определения новых типов: язык не включает в себя конструкции для определения классов, аналогичных Java-классам. Данное свойство языка Ajl обусловлено тем, что используемая модель вычислений основана на парадигме функционального программирования и не включает в себя в полном объёме понятий, характерных для объектно-ориентированного программирования.

1.3. Структура программы

Как было указано в разделе 1.1 при описании используемой модели вычислений, на верхнем уровне структуры программа состоит из T-функций. в то же время, из тела T-функции могут быть вызваны как другие T-функции, так и «обычные», регулярные функции (*методы* в терминологии языка Java)[2]. Таким образом, файл с программой на языке Ajl (*Ajl-модуль*) содержит последовательность определений T-функций и регулярных функций, написанных на языке Java.

Ajl-модуль может содержать раздел импорта в стандартной для языка Java форме, этот раздел — в случае его наличия — должен располагаться до первого определения T-функции или Java-метода в модуле. Непосредственно перед первым определением T-функции или Java-метода должно располагаться определение имени модуля, в соответствии со следующим грамматическим правилом:

`'module'` <*имя модуля*> `';'`

Исполнение программы на языке Ajl начинается с исполнения T-функции tMain.

1.4. Определение T-функции

Определение T-функции, как и определение регулярной функции (Java-метода) состоит из заголовка функции и ее тела в фигурных скобках.

[2]См. раздел 1.4.

Приведем соответствующее грамматическое правило:

```
'[' <описание результатов> ']' <идентификатор> '('
    <описание аргументов> ')' '{'
            <тело T-функции>
'}'
```

Тело T-функции представляет собой последовательность операторов языка Ajl; в нем могут присутствовать операторы, допустимые в теле Java-метода. в частности, возможно создание объектов импортированных классов с использованием оператора `new` (но, что естественно, отсутствуют анонимные классы), также возможно вызывать статические и виртуальные методы объектов — вновь созданных или переданных в T-функцию в качестве входных аргументов. Кроме того в теле T-функции могут быть использованы следующие операторы, специфические для языка Ajl:

- определение T-переменной, см. раздел 1.4.2;
- вызов T-функции («сетевой вызов»), см. раздел 1.4.3;
- рассылка результата (оператор `<-=`), см. раздел 1.4.4.

1.4.1. *Заголовок T-функции*

В соответствии с грамматическим правилом, приведенным выше в разделе 1.4, заголовок T-функции включает описание результатов (возможно, нескольких), аналогично обычному для языка Java описанию аргументов функции, но в квадратных скобках.

Пример:

Ajl
```
1 [Integer i_res] tMain (String [] args)
```

Здесь приведен заголовок T-функции `tMain`, с исполнения которой начинается исполнение всей программы. Эта функция должна иметь один результат типа `Integer` и один аргумент типа «массив строк».

T-функция должна иметь хотя бы один результат, но (как и регулярная функция) может не иметь ни одного аргумента. Аргументы и результаты T-функции являются T-переменными (могут содержать неготовые значения).

1.4.2. *Определение T-переменной*

T-переменные (или, что то же самое, внешние переменные) определяются так же, как и обычные переменные. Единственное отличие заключается в том, что к спецификации типа добавляется описатель `outer`. Пример:

Ajl
```
1 outer Integer oi;
2 outer String [] osa;
```

В приведенном примере определяются две T-переменных:

- внешняя переменная `oi` типа `Integer`;
- массив `osa` внешних значений типа `String[]`. Сама переменная `osa` является обычной («не-T», *внутренней*), и не может содержать неготовые значения, но после индексации (например: `osa[5]`) результатом является ссылка на внешнее значение (которое может быть неготовым).

T-переменные не могут иметь тип, являющийся базовым типом языка Java (таким, как `int`, `float`, `char` и т.п.), их тип должен быть ссылочным и являться подтипом типа Object. Это не является сколь-либо существенным ограничением, поскольку у каждого базового типа есть соответствующий ему тип-обертка (в случае переменной `oi` из примера выше это ссылочный тип-обертка `Integer`, соответствующий базовому типу `int`). При необходимости — например при передаче в регулярный Java-метод — может производиться автоматическое преобразование ссылочного типа к базовому, и наоборот.

T-переменные могут быть определены только в теле T-функции, это существенное свойство T-переменных. При передаче в качестве аргумента в обычные — «не-T» — функции (методы) из T-переменной по общему правилу автоматически извлекается ее значение, которое и передается в качестве значения аргумента вызываемой регулярной функции.

1.4.3. *Вызов T-функции*

В отличие от вызова регулярной функции (Java-метода), который является выражением и возвращает значение того или иного типа, вызов T-функции в Ajl является оператором. T-функция может

возвращать несколько результатов. Синтаксис вызова Т-функции может быть описан следующим грамматическим правилом:

'[' <*список результатов*> ']' '='
    <*идентификатор функции*> '('
        <*список аргументов*> ')' ';'

Пример:

```Ajl
[iHigh, iLow] = bounds (iList);
```

В приведенном здесь вызове гипотетическая Т-функция `bounds` возвращает в Т-переменные `iHigh` и `iLow` соответственно наибольший и наименьший элементы списка целых чисел `iList`. В соответствии с используемой вычислительной моделью исполнение этого оператора приведет к созданию нового Т-процесса, имеющего два выхода и один вход.

Первый выход вновь созданного Т-процесса станет поставщиком значения Т-переменной `iHigh`. Аналогично, второй выход данного Т-процесса станет поставщиком значения Т-переменной `iLow`. На вход Т-процесс получит (потенциально неготовое) значение Т-переменной `iList`. Исполняемый код вновь созданного Т-процесса соответствует коду Т-функции `bounds`. Непосредственно по завершении исполнения приведенного в данном примере оператора переменные `iHigh` и `iLow` будут иметь неготовые значения, а новый Т-процесс окажется в очереди Т-процессов, готовых к исполнению.

1.4.4. *Рассылка результата*

Как уже указывалось в разделах 1.4.1 и 1.4.3, в языке Ajl Т-функция может возвращать несколько результатов. Оператор рассылки (возврата) результата, выполняемый в теле некоторой Т-функции, ставит в соответствие одному конкретному результату этой Т-функции некоторое конкретное значение.

Для описания синтаксиса оператора рассылки результата можно привести следующее грамматическое правило:

<*Идентификатор результата*> '<-=' <*выражение*> ';'

Поскольку неготовое значение, содержащееся в Т-переменной, при присваивании в другую Т-переменную копируется, выход Т-процесса, соответствующий указанному в операторе результату Т-функции, может оказаться поставщиком значения для *нескольких* Т-переменных.

В момент, когда некоторое значение ставится в соответствие данному результату, это значение должно быть *разослано* (присвоено) всем соответствующим T-переменным-потребителям. Именно по этой причине, оператор, осуществляющий в языке Ajl возврат из T-функции одного из результатов, получил название оператора рассылки результата (или просто рассылки).

Пример:

```Ajl
i_res <-= iHigh;
```

Здесь (см. выше примеры в разделах 1.4.1 и 1.4.3) в качестве результата T-функции tMain рассылается значение T-переменной iHigh.

Оператор рассылки отличается от знакомого нам по языку Java оператора возврата (return) тем, что в результате его выполнения T-функция не теряет управления. Исполнение операторов, следующих за оператором рассылки, продолжается — в том, конечно, случае, если оператор рассылки не был последним оператором в теле T-функции. Очевидно, это дает возможность последовательно выполнить вычисление и рассылку нескольких результатов — для случая T-функции, возвращающей несколько результатов.

Оператор рассылки по каждому из результатов T-функции может быть выполнен только однократно; повторное выполнение оператора рассылки по одному и тому же результату T-функции вызывает, в терминах языка Java, возникновение исключения.

Рассылаемое значение может быть неготовым, например в том случае, если правая часть оператора рассылки представляет собой идентификатор T-переменной и в момент выполнения оператора рассылки значение этой внешней переменной не является вычисленным. В результате выполнения такой рассылки неготового значения поставщик этого вновь разосланного неготового значения (выход некоторого T-процесса) становится поставщиком для всех T-переменных-потребителей значения того результата, по которому выполнялась рассылка.

2. Реализация языка Ajl

В данном разделе мы расскажем об основных решениях, принятых при реализации системы программирования на языке Ajl.

В первом подразделе данного раздела рассказывается о реализации легковесных потоков — струй (англ. *fibers*[3]), использованных в реализации для отображения понятия Т-процесса. Второй, завершающий подраздел текущего раздела содержит сведения о выбранном подходе к реализации транслятора языка Ajl: с этой целью была использована каркасная технология (англ. *framework*) Xtext в среде Eclipse.

2.1. Реализация понятия Т-процесса

Важной задачей, которую необходимо было решить при реализации языка Ajl, являлся поиск разумного и достаточно эффективного способа отобразить в реализации понятие Т-процесса. Т-процесс служит для вычисления соответствующей ему Т-функции; он способен быть созданным, переданным на выполнение или приостановленным.

Одним из очевидных претендентов на роль сущности, лежащей в основе реализации Т-процесса, были потоки (англ. *threads*)[4], поддерживаемые в ядре многих современных ОС. В пользу такого решения свидетельствовал, в частности, и тот факт, что достаточно развитая поддержка системных потоков присутствует в стандартной библиотеке современных реализаций языка Java.

Альтернативой использованию системных потоков был подход, в котором Т-процессы отображались в легковесные потоки — струи. Использование струй (как основы для реализации Т-процессов) имеет свои серьезные преимущества: легковесность струй более соответствует их использованию в качестве базового механизма языка. Как показывает опыт, полученный при реализации ядра Т-системы [**2**], использование струй для реализации понятия Т-процесса может приводить к снижению накладных расходов (и, соответственно, повышению эффективности использования аппаратных средств), а также к увеличению числа Т-процессов, способных одновременно функционировать в рамках реализованного с использованием Т-системы приложения.

В рамках работ по реализации языка Ajl было проведено сравнение данных двух подходов при их использовании на платформе JVM. Достаточно развернутое изложение подробностей реализации этих

[3]Буквальный перевод — волокна. Как волокна являются составными частями нити (англ. *thread*), так и струи входят в состав потоков.

[4]в русскоязычной литературе используются также эквивалентный термин *нити управления* (или даже просто *нити*).

подходов и экспериментов по их сравнению приведено в работе [6]. Здесь же мы позволим себе лишь изложить основные результаты данного исследования.

Экспериментальная реализация T-процессов основывалась на свободно-доступной библиотеке Quasar. Данная библиотека предоставляет абстракцию *течения* (англ. *strand*), являющуюся обобщением понятий потока и струи. Отметим также, что библиотека Quasar оказалась единственной библиотекой для платформы JVM, поддерживающей реализацию струй, изолируемую от реализации акторов, которую удалось обнаружить автору на момент выполнения данного исследования.

В результате экспериментов выяснилось следующее:

- в достаточно широком диапазоне нагрузок, приходящихся на один T-процесс, накладные расходы и времена выполнения всего приложения в целом для случая реализации модельных T-процессов струями отличались от соответствующих значений для случая реализации модельных T-процессов потоками незначительно;
- для случая реализации модельных T-процессов струями максимальное число одновременно функционирующих T-процессов, обеспечиваемое реализацией, (около двух миллионов) значительно превышало аналогичную величину, достигаемую при использовании потоков (порядка 2^{16} или 65536);
- при общих числах модельных T-процессов, одновременно функционирующих в составе приложения, находящихся в диапазоне от 2^{12} (4096) до 2^{16} (65536), вариант реализации приложения, основывающийся на использовании струй, обеспечивал значимо меньшие времена выполнения всего приложения (и, соответственно, большую эффективность), чем вариант, основывавшийся на использовании потоков.

С учетом полученных в ходе данного эксперимента результатов было принято решение об использовании струй и библиотеки Quasar для реализации понятия T-процесса в текущей экспериментальной реализации языка Ajl.

2.2. Реализация транслятора языка Ajl

Еще одной важной — и наиболее трудоемкой из всех задач, которые пришлось решать в ходе реализации языка Ajl, была задача реализации транслятора данного языка. С этой целью была

использована каркасная технология Eclipse TMF [7] (от Text Modelling Framework) — средство метапрограммирования, реализованное в среде Eclipse IDE [8] и имеющего также другое, более часто используемое, название Xtext.

2.2.1. *Xtext — каркасная технология разработки трансляторов в среде Eclipse IDE*

Eclipse IDE является одной из наиболее популярных сред разработки для платформы JVM, распространяемых с исходными текстами. Важным свойством данной среды программирования является возможность реализации ее расширений (плагинов), ориентированных на поддержку разработки программ с использованием конкретного языка программирования (или группы таких языков).

Технология Xtext как раз и является одной из технологий разработки подобных расширений. Основной областью, на которую ориентирована данная технология, является поддержка *предметно-ориентированных языков* (англ. *domain-specific languages, DSL*). Данная технология, как и среда Eclipse IDE, распространяется с исходными текстами.

Xtext объединяет в единый обрабатывающий конвейер ряд программных инструментов и средств для разработки и реализации языков программирования, а также обеспечивает возможности их — программных средств и инструментов — настройки и модификации.

Первым этапом обработки входного файла в Xtext является грамматический разбор и построение дерева абстрактного синтаксиса (англ. *abstract syntax tree, AST*).

Для реализации данного этапа в технологии Xtext используется генератор синтаксических анализаторов со входным языком Xtext. Язык Xtext предназначен для описания грамматик и спецификации процесса построения AST. В процессе генерации синтаксического анализатора спецификация на языке Xtext, в свою очередь, преобразуется во входной файл ANTLR — одного из самых мощных современных генераторов синтаксических анализаторов [9], позволяющих задавать грамматику в форме LL(*).

Кроме того, для генерации базовых структур данных, из которых будут впоследствии строиться AST программ, написанных на входном языке, в Xtext используется технология Eclipse Modelling Framework (EMF) [10], разработанная фирмой IBM. В более общем плане, технология EMF используется в рамках Xtext для реализации

идеологии «преобразования моделей» (англ. *model transformation*). В терминах данной идеологии, при генерации синтаксического анализатора порождается *метамодель языка*, из элементов которой впоследствии будут строиться *модели программ* на данном языке (которые, в более привычной терминологии, будут представлять собой деревья абстрактного синтаксиса данных программ). Дополнительно к средствам, присутствующим в реализации технологии EMF непосредственно, Xtext содержит дополнительный набор программных компонент, ориентированных на обработку структур данных EMF.

После грамматического разбора и построения дерева абстрактного синтаксиса следующим этапом в том варианте обрабатывающего конвейера Xtext, который был использован в реализованном трансляторе языка Ajl, является этап построения т.н. *JVM-модели* входной программы на основе AST. Данный этап существенно опирается на являющуюся компонентом Xtext реализацию Xbase — набора программных инструментов и одноименного языка выражений, близких по синтаксису к выражениям языка Java.

Помимо спецификации элементов AST языка Xbase, EMF-метамодель, входящая в состав компоненты Xbase, включает в себя описание таких понятий, как класс, поле и метод, соответствующих одноименным понятиям платформы JVM. Таким образом, можно утверждать, что JVM-модель некоторой входной программы является представлением данной программы в терминах платформы JVM.

Процесс генерации JVM-модели программы по по ее AST в технологии Xtext реализуется разработчиком транслятора входного языка в виде программного модуля, использующего обращения к предоставляемому компонентой Xbase программному инструментарию.

Следующим после этапа построения JVM-модели является этап разрешения имен и вычисления типов. Его базовая, автоматически сгенерированная реализация опирается на входящую в Xtext универсальную библиотеку алгоритмов разрешения имен и типов для описания пространств имен с вложенными областями видимости (англ. *scoping*), а также на входящий в компонент Xbase набор специализированных программных инструментов и заготовок для представления Java-подобного пространства имен, включающего в себя интерфейсы, классы, поля, методы, а также стандартные для языка Java описания импорта.

За этапом разрешения имен следует этап *верификации* моделей — дерева абстрактного синтаксиса и JVM-модели входной программы —

с использованием предоставляемых Xtext инструментов. На этапе верификации производятся проверки требований к семантике входной программы. Также на данном этапе имеется возможность дополнительной проверки синтаксиса программы, что дает возможность использовать более простые грамматики на этапе спецификации синтаксического анализатора и, в конечном счете, способно сильно упростить его — синтаксического анализатора — разработку.

И, наконец, за этапом верификации следует этап генерации кода (в случае транслятора языка Ajl — кода на языке Java). Готовый кодогенератор в язык Java из JVM-модели, обогащенной (на этапе разрешения имен) информацией о типе подвыражений, также входит в набор компонентов Xtext. Впоследствии из сгенерированной программы может быть получен байткод — как с использованием встроенного в Eclipse IDE компилятора языка Java, так и с помощью стандартного для Oracle JDK/OpenJDK компилятора javac.

Следует отметить, что приведенное в данном разделе описание этапов обработки входной программы с использованием технологии Xtext характерно для пакетного транслятора. Последовательность этапов обработки исходных текстов с использованием встроенного в Eclipse IDE интерактивного редактора-компилятора замкнута в цикл и включает в себя дополнительные этапы — такие, как выдача подсказок, быстрое исправление по шаблонам (англ. *quick fixes*) и т.п.

В текущем разделе мы привели сжатое описание основных компонентов каркасной технологии Xtext. Следующий раздел будет посвящен настройкам, которые позволили модифицировать компоненты Xtext, входящие в состав стандартной для данной технологии последовательности обрабатывающих стадий, для реализации транслятора языка Ajl.

2.2.2. *Настройки компонентов Xtext, использованные в трансляторе Ajl*

Eclipse TMF — Xtext — обладает высоким уровнем настраиваемости, как за счет использования конфигурационных файлов (на этапе автоматической генерации ряда компонентов), так и за счет внедрения в программный код зависимостей с использованием программного комплекса Google Guice. Очевидно, что подобный уровень настраиваемости является необходимым для каркасной технологии метапрограммирования, поскольку за счет фиксированной последовательности стандартных этапов обработки невозможно обеспечить

процесс трансляции входных файлов на языках, обладающих значительным разнообразием синтаксических и семантических свойств.

Заняв крайнюю точку зрения, можно было бы даже считать, что вся разработка транслятора с использованием Eclipse TMF заключается в разработке и реализации настроек для этой технологии на конкретный язык программирования. Конечно, эти «настройки» — в случае, когда реализуемый язык является достаточно развитым — могут включать в себя весьма значительный объем кода на языке Java или других Java-подобных языках, таких, например, как Xtend. С этой точки зрения, реализация языка программирования Ajl с использованием Xtext является развитием набора настроек данной технологии, разработанных Лоренцо Беттини для реализации языка Java−− [11]. Поскольку язык Java−− не включает в себя основные понятия модели вычислений «самотрансформация вычисляемой сети», то объем дополнительных «настроек» технологии Xtext, которые были разработаны в целях реализации языка Ajl, составил весьма значительную величину.

На этапе генерации синтаксического анализатора и метамодели языка настраиваемость заявляет о себе со всей очевидностью, поскольку грамматика, объединенная с алгоритмом построения AST, специфицируется на языке Xtext в соответствующем файле. Описание грамматики (и процесса построения AST) языка Ajl выполнено на основе грамматики Xbase, при этом не только определяются необходимые новые грамматические правила, но и переопределяется ряд грамматических правил Xbase.

Однако неочевидной является возможность использования предопределенного (а не генерируемого по Xtext-файлу) описания метамодели на языке XML — в файле с расширением “`.ecore`”. Именно такой подход, с опорой на наследование от предопределенной метамодели компонента Xbase, был использован для разработки метамодели языка Ajl. С этой целью была использована специализированная графическая среда, предназначенная для редактирования EMF-метамоделей, входящая в состав технологии EMF.

В процессе функционирования транслятора языка Ajl, после того, как выполнил свою работу синтаксический анализатор и было построено AST транслируемой программы, происходит генерация JVM-модели данной программы (см. раздел 2.2.1). В соответствии со стандартной практикой, для этой цели был реализован программный модуль `AJLJvmModelInferrer` на языке Xtend; данный модуль также может рассматриваться как элемент «настроек».

Одной из главных проблем, которые разработчикам пришлось решать при реализации модуля `AJLJvmModelInferrer`, было отсутствие в составе метамодели используемого в Xtext JVM-представления таких элементов, которые соответствовали понятиям T-переменной, T-процесса, операциям вызова T-функции, рассылки результата и других понятий используемой модели вычислений. В результате было принято решение о том, что примитивы модели вычислений следует отображать в близкие по смыслу элементы JVM-модели:

- определение T-функции — в определение метода, не возвращающего результата (возвращающего `void`). При этом результаты T-функции отображаются в начальные элементы списка аргументов соответствующего метода;
- вызов T-функции — в вызов соответствующего метода;
- определение T-переменной — в определение обычной (внутренней) переменной;
- оператор рассылки результата — в присваивание.

Следует отметить, что предоставляемый компонентом Xbase инструментарий для построения JVM-модели позволяет устанавливать связь между элементом данной модели и элементом AST, на основе которого этот элемент JVM-модели был создан. За счет данной связи на следующих этапах обработки можно будет, при необходимости, восполнить кажущуюся потерю информации, возникающую при изложенном выше способе отображения понятий модели вычислений в элементы JVM-модели исходной программы.

За счет того, что в JVM-модели исходной программы нет новых вариантов элементов (по сравнению с реализацией языка Java−−), на этапе разрешения имен и вычисления типов не требуется дополнительных действий по настройке каркасной технологии Xtext.

В противоположность этому, на этапе верификации моделей был реализован значительный объем разнообразных дополнительных проверок. В качестве примера можно привести проверку того, что в левой части оператора рассылки результата расположен идентификатор результата T-функции. Другим примером может служит проверка того факта, что определение T-переменной находится в теле T-функции. Очевидно, что для этой цели необходимо использовать информацию, отсутствующую в JVM-модели входной программы, поэтому используются программные инструменты, позволяющие по элементу JVM-модели получить доступ к соответствующему элементу AST.

И, наконец, на этап генерации кода приходится наибольший объем настроек. Поэтому остановимся на данном этапе обработки входной программы подробнее.

С целью настройки этапа генерации кода используется механизм Google Guice:

- на место класса `XbaseCompiler`, предоставляемого компонентом Xbase, инжектируется реализованный автором на языке Java класс `AjlXbaseCompiler`, являющийся наследником класса `XbaseCompiler`;
- на место интерфейса `IGenerator` инжектируется реализованный автором на языке Xtend класс `AjlJvmModelGenerator`, являющийся наследником класса `JvmModelGenerator`, также предоставляемого компонентом Xbase.

Как результат, в качестве генератора кода используется стандартный для компонента Xbase генератор кода, в котором часть методов оказывается переопределенной.

При обработке верхнего уровня структуры JVM-модели из стандартного генератора кода в процессе итерации по элементам класса — а класс в JVM-модели соответствует модулю входной программы на языке Ajl — вызывается метод `generateMember` инжетированного класса `AjlJvmModelGenerator`. В данном методе за счет доступа к соответствующему элементу AST определяется, не является ли обрабатываемый элемент JVM-модели образом T-функции. В этом случае вызывается специализированный метод, ответственный за генерацию вложенного класса, соответствующего данной T-функции — поскольку образом каждой T-функции, в соответствии с интерфейсом библиотеки Quasar, является отдельный класс.

Аналогичным образом в инжектированном классе `AjlXbaseCompiler` переопределяются методы класса `XbaseCompiler`, что дает возможность корректно оттранслировать тело T-функции в код метода `run` (на языке Java) того вложенного класса, который соответствует данной T-функции. В частности, таким образом обеспечивается трансляция определения T-переменной, извлечения значения такой переменной, вызова T-функции (сетевого вызова), рассылки результата в теле T-функции.

В данном разделе обсуждались различные аспекты выполненной реализации языка Ajl. Следующий раздел будет посвящен описанию эксперимента, поставленного с целью изучения свойств данной реализации, а также изложению и обсуждению результатов, полученных в этом эксперименте.

3. Экспериментальное изучение выполненной реализации языка Ajl

По завершении разработки работоспособной версия пакетного транслятора с языка Ajl на язык Java перед разработчиками возникла задача оценки качества выполненной реализации. В частности, было необходимо выяснить, обеспечивает ли данная реализация эффективное использование вычислительной мощности современных архитектур широкого применения, выполненных с использованием многоядерных микропроцессоров. Текущий раздел публикации посвящен описанию эксперимента, поставленного с целью получения обоснованного ответа на этот вопрос.

3.1. Постановка эксперимента

Для оценки свойств реализации языка Ajl на данном языке была реализована модельная программа, имитирующая решение задачи с потенциально массовым параллелизмом. Решение такой задачи для некоторой области вычисляется по следующей схеме:

- исходная область разбивается на несколько приблизительно равных фрагментов;
- для каждого из фрагментов решается задача одного и того же типа, решения для различных фрагментов могут вычисляться параллельно;
- результаты, полученные при решении задач для отдельных фрагментов используются для вычисления их обобщенной суммы, которая и выдается в качестве результата.

В качестве одного из наиболее известных примеров задачи такого типа можно привести широко известную задачу построения псевдореалистичного изображения методом трассировки лучей: в этом случае исходной областью является прямоугольник, в котором строится изображение. Этот прямоугольник разбивается на прямоугольники меньшего размера, затем для каждого из полученных в результате такого разбиения прямоугольников производится построение изображения стандартным вариантом метода трассировки лучей. Наконец, общее изображение собирается из построенных прямоугольных фрагментов, этот процесс сборки целого изображения из фрагментов и можно рассматривать как вариант обобщенного суммирования.

Для проведения эксперимента в качестве модельной задачи использовалась процедура интенсивного тестирования памяти.

Ajl

```
1  [Object res] bTree (Integer cnt, Integer csize,
2                      Integer minsize) {
3    int halfsize = csize / 2;
4    if (halfsize < minsize) {
5      outer Integer oi;
6      [oi] = leaf (cnt, csize);
7      res <-= oi;
8    } else {
9      outer Object [] node = new outer Object [2];
10     [node [0]] = bTree (cnt, halfsize, minsize);
11     [node [1]] = bTree (cnt, halfsize, minsize);
12     res <-= node;
13   }
14 }
15
16 [Integer res] leaf (Integer cnt, Integer size) {
17   res <-= StreamMemTest.newstream (cnt, size) ;
18 }
```

Рис. 1. T-функции bTree и leaf

Решение задачи с потенциально массовым параллелизмом в экспериментальном приложении было реализовано с использованием шаблона проектирования «fork-join параллелизм». В своей рекурсивной инкарнации данный шаблон проектирования опирается на парадигму «разделяй и властвуй» (см., например, [**12**]).

На рис. 1 приведен текст на языке Ajl основной части программы, использованной в эксперименте.

Назначение T-функции `bTree` (строки 1–14) — рекурсивное деление области памяти на равные фрагменты. На вход ей подается 3 аргумента: число повторений теста памяти (`cnt`), а также текущий (`csize`) и минимальный (`minsize`) размеры фрагмента.

В строке 3 вычисляется величина `halfsize`, равная половине текущего размера фрагмента.

В строке 4 этот половинный размер сравнивается с минимальным, и если `halfsize` оказывается меньше минимального размера фрагмента, то процесс рекурсивного деления считается завершенным, вызывается базовая («листовая») T-функция `leaf`, которой передаются число повторений теста памяти и текущий размер фрагмента. В строке 7 результат этого вызова T-функции `leaf` возвращается в качестве результата T-функции `btree`.

Если же условие проверки в строке 4 не выполняется и половинный размер оказывается равным минимальному либо превышает его, то процесс рекурсивного деления продолжается. Создаются два новых узла бинарного дерева (строка 9), которые сохраняются на концах двух дуг, исходящих из узла `node` — иначе выражаясь, узлы дерева являются элементами вновь созданного массива `node`. Создаются (строки 10–11) два рекурсивных вызова T-функции `btree`, которым вместо текущего размера памяти передается половинный (`halfsize`), результат каждого из этих двух вызовов сохраняется в соответствующем вновь созданном (в строке 9) узле бинарного дерева. В качестве результата текущего вызова `btree` в данном случае возвращается (строка 12) узел бинарного дерева `node`, содержащий дуги, на концах которых хранятся вновь созданные узлы бинарного дерева.

С операционной точки зрения, T-функция `btree` реализует процесс построения бинарного дерева, в листьях которого находятся результаты вызовов T-функции `leaf`. Построение ветвей, исходящих из одного узла, может, потенциально, производиться параллельно друг другу — впрочем, как и вычисление результатов, находящихся в листьях.

T-функция leaf (строки 16–18) вызывает метод `newstream` класса StreamMemTest (исходный код выполнен на языке Java, находится в отдельном файле). Метод `newstream` реализует перенесенную с языка C на язык Java процедуру интенсивного тестирования памяти, входящую в пакет Stream [**13**, **14**], предназначенный для измерения пропускной способности оперативной памяти компьютера.

После запуска решения модельной задачи для фрагментов начальной области следует этап вычисления обобщенной суммы на основе всех результатов решения задачи для каждого из данных фрагментов; исполнение данного этапа обеспечивает T-функция `traverse`, текст которой приводится на рис. 2. Данная T-функция реализует обход построенного на предыдущем этапе двоичного дерева, в листьях которого находятся потенциально неготовые результаты вычисления отдельных фрагментов задачи.

В строке 2 из T-переменной узла извлекается содержащееся в нем значение и проверяется, не имеет ли это значение тип `Integer`. Если условие проверки истинно, то переданный узел является листом, и в нем содержится результат вычисления фрагмента задачи (в нашем случае — всегда 1). Этот результат и возвращается в качестве результата вызова T-функции `traverse`.

Ajl

```
1  [Integer res] traverse (Object node) {
2    if (node instanceof Integer) {
3      res <-= (Integer) node;
4    } else {
5      int cnt = 0;
6      outer Object [] noda = (outer Object []) node;
7      for (int i = 0; i <= 1; i++ ) {
8        outer Integer oi;
9
10       [oi] = traverse (noda[i]);
11       cnt += oi;
12     }
13     res <-= cnt;
14   }
15 }
```

Рис. 2. T-функция traverse

Если же условие проверки в строке 2 не выполняется, это означает, что необходимо последовательно обойти оба поддерева, исходящих из данного узла и сложить возвращенные в результате результаты (строки 5–14). В строке 6 извлекается значение из переданной на вход T-функции `traverse` T-переменной `node` (аргумент T-функции всегда является T-переменной), после чего извлеченное значение преобразуется к типу массив T-переменных типа Object — данное значение затем сохраняется в переменной `noda`. Далее (строки 7–12) в цикле из двух итераций производится рекурсивный вызов (строка 10) T-функции `traverse` для каждого из двух поддеревьев. Полученные от рекурсивных вызовов значения накапливаются (строка 11) в переменной `cnt`. Значение, накопленное в переменной `cnt` возвращается (строка 13) в качестве результата данного T-процесса, вычисляющего T-функцию `traverse`.

Наконец, на рис. 3 приведен текст T-функции `tMain`, начальной T-функции всей использованной для проведения эксперимента программы на языке Ajl.

Выполнение данной T-функции начинается (строки 2–5) с присваивания умолчательных значений переменным `cnt` (содержит число повторений теста памяти), `size` (содержит размер памяти в терминах числа переменных типа double) и `minsize` (содержит ограничение снизу на число переменных в одном фрагменте). Затем (строки 6–14) данные значения корректируются с учетом переданных программе — в виде

```
1  [Integer i_res] tMain (String [] args) {
2    Integer
3      cnt = 1,
4      size = 8388608,
5      minsize = 1024;
6    if (args.length > 0) {
7      cnt = Integer.parseInt (args[0]);
8    }
9    if (args.length > 1) {
10     size = Integer.parseInt (args[1]);
11   }
12   if (args.length > 2) {
13     minsize = Integer.parseInt (args[2]);
14   }
15   outer Object ooa;
16   [ooa] = bTree (cnt, size, minsize);
17   outer Integer oi;
18   [oi] = traverse (ooa);
19   System.out.println(oi);
20   i_res <-= 0;
21 }
```

Рис. 3. T-функция tMain

текстовых строк — входных параметров.

После этого (строка 16) вызывается — обсуждавшаяся выше — T-функция `btree`, которой значения переменных `cnt`, `size` и `minsize` передаются в качестве аргументов. Результат (возможно, неготовый) вызова T-функции `bTree` передается в качестве аргумента вызову T-функции `traverse` (строка 18). Заметим, что процессы построения двоичного дерева в вызове T-функции `bTree` и обхода этого дерева в вызове T-функции `traverse` могут выполняться одновременно (параллельно).

Наконец (строка 19), из результата вызова функции `traverse`, хранящегося в T-переменной `oi`, извлекается его значение и выдается в виде строки символов на стандартный вывод программы. В соответствии с описанной логикой программы, это значение равно общему числу фрагментов, на которое была разбита область в процессе деления. В строке 20 в качестве результата работы программы возвращается целое значение 0, что означает, что программа выполнилась корректно.

Как видно из приведенного описания программы, областью, подлежащей делению, был общий объем тестируемой памяти. В экспериментальных запусках этот объем составлял 8388608 чисел с плавающей точкой базового типа double, каждое размером 8 байтов, что составляет 64 Мбайт оперативной памяти. В результате деления данной области на фрагменты каждому запуску соответствовало кратное степени 2 число равных фрагментов. Число повторений теста подбиралось таким образом, чтобы при последовательном исполнении программы (всего один фрагмент размером 64 Мбайт и, как следствие, один T-процесс, соответствующий вызову T-функции `leaf`, реализующей тестирование памяти) время исполнения приблизительно составляло 1 мин.

Эксперимент проводился на ЭВМ, оснащенной ОЗУ объемом 16 Гбайт и 4-ядерным процессором Intel® Core™i7 930, работающим на частоте 2.80 ГГц. ЭВМ в ходе эксперимента функционировала под управлением 64-битной ОС Linux с ядром версии 3.16.36. Для исполнения приложения использовалась 64-битная виртуальная машина языка Java версии 7u111 из состава дистрибутива OpenJDK, использующая JIT-компилятор HotSpot.

Программа запускалась в двух режимах:

- регулярный режим, когда в ходе исполнения программы функционировал JIT-компилятор (при этом отдельные участки программы могли быть скомпилированы в машинный код);
- режим интерпретации, когда в ходе исполнения компиляция в машинный код не выполнялась (такой режим исполнения программы задает параметр `-Xint` виртуальной машины Java).

3.2. Результаты эксперимента

Результаты эксперимента приведены на графике (см. рис. 4).

По горизонтальной оси на данном графике в логарифмическом масштабе откладывается общее число фрагментов, на которые в экспериментальном запуске была разделена область. Например, значение 0 соответствует одному фрагменту ($2^0 = 1$), значение 10 — 1024 фрагментам ($2^{10} = 1024$).

По вертикальной оси откладывается коэффициент ускорения, то есть отношение времени исполнения для случая, когда распараллеливание не выполнялось (вся область состояла из одного фрагмента, т.е. тестировался всего один участок памяти с размером, равным размеру всей области) к времени исполнения соответствующего

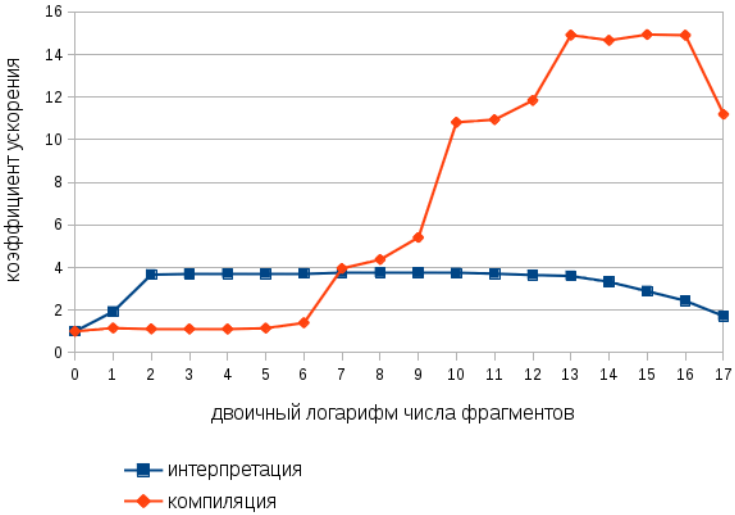

Рис. 4. Экспериментальный график зависимости коэффициента ускорения от общего числа фрагментов

экспериментального запуска. Например, для значения абсциссы, равного 0, что соответствует области, состоящей из одного фрагмента, коэффициент ускорения, очевидно, равен 1.

Рассмотрим сначала случай регулярного исполнения, когда JIT-компилятор отключен не был (соответствующая ломаная отмечена в легенде на графике надписью «компиляция»).

3.2.1. *Случай регулярного исполнения («компиляция»)*

В этом случае для запусков, в которых число фрагментов находилось в диапазоне от 1 (последовательное исполнение) до 32 (2^5), коэффициент ускорения, как следует из графика, был близок к 1. В чем причина такого поведения, ведь если основная часть времени исполнения тратится на обработку фрагментов, и каждый из фрагментов обрабатывается отдельным T-процессом, то из-за наличия в четырехядерном процессоре возможности одновременно выполнять 4 струи в четырех системных потоках с увеличением числа фрагментов время исполнения должно сокращаться, а коэффициент ускорения — стремиться к четырем?

Фактором, сдерживающим рост коэффициента ускорения, является в данном случае ограниченная пропускная способность оперативной памяти. Внутренний цикл процедуры тестирования памяти был эффективно скомпилирован в машинный код, и даже одна струя, выполняющая данный код в рамках одного системного потока, практически полностью использует всю пропускную способность памяти. Поэтому увеличение числа одновременно исполняющихся струй (растущее до четырех с ростом числа фрагментов) приводит к пропорциональному увеличению задержки обращения к памяти, исполнение каждой из струй замедляется, и в результате общее время исполнения приложения не сокращается.

Когда число фрагментов находится в диапазоне от 64 (2^6) до 256 (2^8), по мере нарастания числа фрагментов наблюдается интенсивный рост коэффициента ускорения — в пределах от 1.4 до 4.4. Объяснением такого ускорения работы приложения является начало эффективной работы процессорного кэша 3-го уровня, который разделяется всеми четырьмя ядрами процессора. Повышение же эффективности работы кэша 3-го уровня связано, в свою очередь, с тем, что с ростом числа фрагментов их размер уменьшается, и объем памяти, соответствующий одному фрагменту, уже полностью умещается в кэш. Отметим, что уже для числа фрагментов, равных 2^7, коэффициент ускорения равен четырем, то есть эффективность работы кэша 3-го уровня выросла настолько, что конкуренция за возможность обращения к оперативной памяти из различных T-процессов практически была сведена к нулю.

Аналогичным образом, интенсивный рост коэффициента ускорения (в пределах 5.4–10.9) в диапазоне числа фрагментов от 512 (2^9) до 2048 (2^{11}) может быть объяснен началом эффективной работы кэша 2-го уровня (кэш 2-го уровня является собственным, отдельным для каждого из ядер). А началом эффективной работы кэша 1-го уровня (32 Кбайт в каждом из ядер) объясняется рост коэффициента ускорения (в пределах 11.8–14.9) в тех запусках экспериментальной программы, в которых число фрагментов составляло от 4096 (2^{12}) до 8192 (2^{13}).

На конечном участке графика, соответствующего числу фрагментов от 2^{14} до 2^{17}, рост коэффициента ускорения (за счет работы кэша 1-го уровня, а также внутренних буферов и регистров процессора) нивелируется интенсивным ростом — в процентном отношении — накладных расходов, поскольку объем полезной вычислительной

нагрузки в каждом из T-процессов, обрабатывающих отдельный фрагмент области, уменьшается, а объем накладных расходов при этом сохраняется. В конце концов тенденция к замедлению берет верх, и коэффициент ускорения начинает падать.

Перейдем теперь к рассмотрению результатов экспериментальных запусков приложения в альтернативном режиме, когда виртуальной машине JVM передавался аргумент `-Xint`, вследствие чего JIT-компилятор во время исполнения приложения не использовался.

3.2.2. *Случай интерпретации (JIT-компилятор отключен)*

В данном случае основная часть вычислительной мощности расходуется в коде JVM: при исполнении каждой JVM-инструкции после ее выборки из памяти виртуальная машина Java производила ее анализ и вызов соответствующей процедуры, реализующей исполнение данной инструкции (на рис. 4 соответствующая ломаная отмечена в легенде надписью “интерпретация”). Пропускная способность памяти в таком режиме исполнения используется не полностью. Таким образом, при данном методе запуска экспериментального приложения моделируется исполнения тех задач, для которых скорость доступа к оперативной памяти не является критичной.

Вследствие этого, в диапазоне значений числа фрагментов от 1 (последовательное исполнение) до 4 (2^2) наблюдается классический линейный рост коэффициента ускорения от 1 до 3.66 за счет роста числа задействованных ядер процессора.

На том участке графика, на котором число фрагментов составляет от 8 (2^3) до 8192 (2^{13}), коэффициент ускорения остается примерно постоянным (в районе 3.7), поскольку дополнительную вычислительную мощность привлечь неоткуда: все 4 ядра процессора уже задействованы.

Наконец, тогда, когда число фрагментов находится в диапазоне от 2^{14} до 2^{17}, за счет интенсивного роста (в процентном отношении) накладных расходов — так же, как и в том случае, когда JIT-компилятор отключен не был (см. раздел 3.2.1) — коэффициент ускорения последовательно уменьшается вплоть до значения 1.7.

3.3. Выводы по результатам эксперимента

На основе анализа результатов экспериментальных прогонов модельного приложения можно сделать следующие заключения:

- выполненная разработка языка Ajl обеспечивает достаточно высокую эффективность реализации вычислительной модели «самотрансформация вычисляемой сети»;
- использованный в реализации модельного приложения шаблон проектирования «fork-join-параллелизм» в его рекурсивном варианте, обеспечивающем реализацию стратегии «разделяй и властвуй», адекватен структуре задачи с потенциально массовым параллелизмом.

Действительно, для разных вариантов приложения — как для варианта, связанного с интенсивными обращениями к оперативной памяти, так и для варианта, где оперативная память не являлась критическим ресурсом — удалось обеспечить высокие значения коэффициента ускорения, при оптимальном размере гранулы параллелизма.

Для второго случая (память — не критический ресурс) достижение коэффициента ускорения, близкого к теоретическому пределу, равному числу ядер процессора, было вполне ожидаемым. В то же время, возможность значительного ускорения приложения для случая, когда JIT-компилятор не отключен (вариант «компиляция») и подсистема оперативной памяти компьютера подвергается интенсивным нагрузкам, стала приятным сюрпризом для исследователей; данный результат продемонстрировал, что выполненная реализация позволяет эффективно использовать доступные возможности аппаратных средств в целях эффективного распараллеливания и ускорения выполнения программы.

Также следует отметить важность правильного выбора такого параметра, как размер гранулы параллелизма. С одной стороны, в варианте интенсивных обменов с оперативной памятью наивысшего значения коэффициента ускорения удалось добиться в том случае, когда размер отдельного фрагмента (области памяти), с которым оперирует «листовой» T-процесс, соответствует размеру кэша первого уровня процессора. С другой стороны, для того же размера фрагмента (или, что то же самое, для того же общего числа фрагментов, равного 2^{13}), коэффициент ускорения близок к оптимальному и для того варианта, когда обмены с памятью не были сдерживающим фактором. Таким образом, в качестве дополнительного критерия при выборе размера гранулы параллелизма целесообразно использовать размер активно используемой области памяти.

4. Близкие работы

Помимо уже упомянутых работ по реализации различных версий Т-системы (см. [1–5]) и соответствующих входных языков, можно выделить следующие основные категории близких работ: современные языки программирования и реализации парадигм параллельного программирования.

Среди *современных языков программирования* следует выделить язык программирования Go[5] [15–17], реализация которого была выполнена командой разработчиков фирмы Google.

Go содержит конструкцию, которая позволяет запустить исполнение вызова некоторой функции в отдельной струе. Такая функция называется в Go горутиной (англ. *goroutine*), ниже приведен пример оператора вызова горутины:

Go

```
1 go f(args)
```

Другая концепция Go, близкая к используемым в Ajl, — концепция канала (англ. *channel*) — предназначена для передачи данных между горутинами и позволяет обеспечить синхронизацию в процессе их выполнения. Использование каналов (или переменных типа `Future`, реализованных на их основе) позволяет реализовать на языке Go приложения в рамках вычислительной модели «самотрансформация вычисляемой сети».

Следует отметить, что для случая языка Go — в отличие от языка Ajl — преобразование последовательной программы к форме, которая реализует данную вычислительную модель, требует внесения в текст программы значительных изменений, чреватых ошибками, поэтому реализуемую таким образом парадигму распараллеливания трудно назвать «автоматическим распараллеливанием». Еще одним отличием языка Go от Ajl является тот факт, что язык Go непосредственно компилируется в машинный код, в то время, как для исполнения Ajl-программы требуется виртуальная JVM-машина.

Язык DPJ (Deterministic Parallel Java, см. [18, 19]) предполагает автоматический контроль корректности распараллеливания, для того чтобы обеспечить детерминированность результата работы параллельного приложения. С этой целью в работах по DPJ активно развивается техника, основанная на специальной *системе типов и*

[5]Данный язык также известен под наименованием Golang.

эффектов (англ. *type and effect system*), для краткости часто именуемой *системой эффектов* (англ. *effect system*). DPJ, как и Ajl, реализуется для платформы JVM. Данный язык строится как расширения языка Java предложениями параллельного исполнения `cobegin`, а также специального вида аннотациями эффекта и декларациями *участков памяти* (англ. *regions*).

Перенос программы с Java на DPJ достаточно трудоемок, в то же время использование системы эффектов и типов обеспечивает контроль ошибок распараллеливания во время компиляции. С другой стороны, язык DPJ, в отличие от Ajl, полностью включает все объектно-ориентированные возможности Java.

Политиповую библиотеку Scalan [**20**] можно отнести скорее к классу *реализаций парадигм параллельного программирования*, в данном случае — так называемого *вложенного параллелизма по данным* [**21**] (англ. *nested data parallelism*).

Scalan, как и Ajl, в качестве базовой платформы использует JVM, но, в отличие от Ajl, в качестве базового языка использует язык Scala. С использованием возможностей языка Scala данная библиотека предоставляет интерфейс в виде встроенного (англ. *embedded*) предметно-ориентированного языка.

Наконец, к тому же классу реализаций парадигм параллельного программирования можно (условно) отнести и библиотеку Akka [**22**, **23**]. Данная библиотека реализует вычислительную модель на основе понятия актора [**24**]. Вообще говоря, модель акторов — универсальная вычислительная модель, которая имеет более широкую область применения, чем разработка параллельных программ. В то же время, данной модели вычислений, несомненно, присущ внутренний параллелизм, что активно используется на практике. Базовым языком, для которого предназначена данная библиотека, является язык программирования Scala, но Akka также предоставляет и интерфейс для ее использования из программ, написанных на Java.

Вычислительная модель акторов радикально отличается от базовой для языка Java объектно-ориентированной модели вычислений; данное отличие приводит к тому, что при использовании библиотеки Akka для реализации параллельных приложений их приходится радикально перерабатывать — в отличие от Ajl, где параллельность может быть внесена небольшим числом не слишком значительных изменений исходных текстов приложения.

Заключение

Нынешнее состояние программного обеспечения и аппаратных средств широкого применения настойчиво требует развития инструментов параллельного программирования на основе языка Java.

В данной публикации рассмотрены различные аспекты выполненной реализации языка Ajl, построенного на основе языка Java и предоставляющего возможность использования парадигмы программирования «автоматическое динамическое распараллеливание». Описаны синтаксис и семантика расширяющих Java конструкций языка Ajl, а также использованный подход к реализации понятия Т-процесса и методы создания транслятора Ajl.

Приведены результаты эксперимента, направленного на изучение свойств выполненной разработки, свидетельствующие о том, что данная реализация языка Ajl обеспечивает достаточно высокую эффективность функционирования приложений на основе использования вычислительной модели «самотрансформация вычисляемой сети».

Таким образом, то направление по развитию инструментария для платформы JVM, в рамках которого была выполнена реализация языка Ajl, можно считать перспективным, а саму реализацию — заслуживающей работы по дальнейшему ее развитию.

Список литературы

[1] S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Yu. V. Shevchuck. «Multiprocessor with automatic dynamic parallelizing», *Transputer: Research and Application*, NATUG 6-th Meeting, IOS Press, Vancouver, 1993, с. 333–344. ↑ 84,110

[2] С. М. Абрамов, А. И. Адамович, М. Р. Коваленко. «Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей», *Программирование*, **25**:2 (1999), с. 100–107. ↑ 84,92,110

[3] С. М. Абрамов, В. А. Васенин, Е. Е. Мамчиц, В. А. Роганов, А. Ф. Слепухин. «Динамическое распараллеливание программ на базе параллельной редукции графов. Архитектура программного обеспечения новой версии Т-системы», *Высокопроизводительные вычисления и их приложения*, Труды Всероссийской научной конференции (30 октября–2 ноября 2000 г., г. Черноголовка), Изд-во МГУ, М., 2000, с. 261–264. ↑ 84,110

[4] A. Moskovsky, V. Roganov, S. Abramov. «Parallelism granules aggregation with the T-system», *Proceedings of the 9th international conference on Parallel Computing Technologies*, PaCT'07 (Pereslavl-Zalessky, Russia, September 3–7, 2007), Lecture Notes in Computer Science, т. **4671**, Springer-Verlag, Berlin–Heidelberg, 2007, с. 293–302. ↑ 84,110

[5] В. А. Роганов, А. А. Кузнецов, Г. А. Матвеев, В. И. Осипов. «Методы адаптации системы параллельного программирования OpenTS для поддержки работы T-приложений на гибридных вычислительных кластерах», *Программные системы: теория и приложения*, **4**:4(18) (2013), с. 17–31, URL: http://psta.psiras.ru/read/psta2013_4_17-31.pdf ↑ 84,110

[6] А. И. Адамович. «Струи как основа реализации понятия T-процесса для платформы JVM», *Программные системы: теория и приложения*, **6**:4(27) (2015), с. 177–195, URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf ↑ 93

[7] Eclipse Text Modelling Framework (Xtext), URL: http://www.eclipse.org/Xtext ↑ 94

[8] Eclipse IDE, URL: http://www.eclipse.org ↑ 94

[9] T. Parr. *ANTLR (ANother Tool for Language Recognition)*, URL: http://www.antlr.org ↑ 94

[10] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*, 2nd edition, Safari Books Online, 2008, URL: https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885 ↑ 94

[11] L. Bettini, Java−−, URL: https://github.com/LorenzoBettini/javamm ↑ 97

[12] Т. Кормен, Ч. Лейзерсон, Р. Ривест. *Алгоритмы: построение и анализ*, МЦМНО, М., 1999, ISBN: 5-900916-37-5, 960 с. ↑ 101

[13] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, A continually updated technical report, University of Virginia, Charlottesville, Virginia, 1991–2007, URL: http://www.cs.virginia.edu/stream/ ↑ 102

[14] J. Hammond. *STREAM*, URL: https://github.com/jeffhammond/STREAM/tree/master ↑ 102

[15] *The Go Programming Language Documentation*, URL: https://golang.org/doc/ ↑ 110

[16] S. Ajmani. «Program your next server in Go», *Proceedings of Applicative 2016* (New York University, June 1st–2nd, 2016), ACM, New York, USA, URL: https://talks.golang.org/2016/applicative.slide#1 ↑ 110

[17] C. Doxsey. *Introducing Go*, O'Reilly Media, 2016, ISBN: 978-1-4919-4195-9, 128 с. ↑ 110

[18] R. L. Bocchino, Jr. , V. S. Adve , S. V. Adve, M. Snir. «Parallel programming must be deterministic by default», *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09 (March 30–31, 2009, Berkeley, California), с. 4-4. ↑ 110

[19] R. Bocchino, et al. «A type and effect system for deterministic parallel Java», *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09 (October 25–29, 2009, Disney's Contemporary Resort, Orlando, Florida), ACM, New York, NY, USA, 2009, с. 97–116, ISBN: 978-1-60558-766-0. ↑ 110

[20] A. V. Slesarenko. «Scalan: polytypic library for nested parallelism in Scala», *Keldysh Institute preprints*, 2011, 022, 28 с., URL: http://library.keldysh.ru/preprint.asp?id=2011-22&lg=e ↑ 111

[21] G. E. Blelloch. «Programming parallel algorithms», *Communications of the ACM*, **39**:3 (March 1996), с. 85–97. ↑ 111

[22] Akka Documentation, URL: http://akka.io/docs/ ↑ 111

[23] M. K. Gupta. *Akka Essentials*, Packt Publishing, Birmingham, UK, 2012, ISBN: 978-1-84951-828-4. ↑ 111

[24] C. Hewitt, P. Bishop, R. Steiger. «A universal modular ACTOR formalism for artificial intelligence», *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73 (August 20–23, 1973, Stanford, California, USA), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, с. 235–245. ↑ 111

Рекомендовал к публикации *к.ф.-м.н. С. А. Романенко*

Пример ссылки на эту публикацию:

А. И. Адамович. «Язык программирования Ajl: автоматическое динамическое распараллеливание для платформы JVM», *Программные системы: теория и приложения*, 2016, **7**:4(31), с. 83–117.

URL: http://psta.psiras.ru/read/psta2016_4_83-117.pdf

Об авторе:

Алексей Игоревич Адамович

Старший научный сотрудник ИПС им. А.К. Айламазяна РАН. Работает в области инструментальных средств для разработки параллельных приложений. Разработчик первой параллельной версии Т-системы, ведущий разработчик параллельного отладчика tdb. Активный участник суперкомпьютерного проекта «СКИФ» Союзного государства России и Беларуси.

e-mail: lexa@adam.botik.ru

Alexei Adamovich. *The Ajl programming language: the automatic dynamic parallelization for the JVM platform.*

Abstract. The current state of the art in the area of the general purpose software and hardware strongly requires the development of tools for parallel programming based on the Java language. In the A. K. Aylamazyan Program Systems Institute of RAS the Ajl programming language was implemented, which is the extension of Java and is designed to facilitate the development of parallel programs with the use of the computation model named "self-transformation of computational network". The paper covers various aspects of the research: syntax and semantics of the language constructs, methods of implementing the translator and the basic concepts of the computation model. The experiments are described that were performed to evaluate the implementation, and their results are discussed. In conclusion, a brief overview of related work is given. (*In Russian*).

Key words and phrases: programming languages implementation, parallel computations, Java programming language, automatic dynamic parallelization, fibers.

References

[1] S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Yu. V. Shevchuck. "Multiprocessor with automatic dynamic parallelizing", *Transputer: Research and Application*, NATUG 6-th Meeting, IOS Press, Vancouver, 1993, pp. 333–344.

[2] S. M. Abramov, A. I. Adamovich, M. R. Kovalenko. "T-System — An Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Ray Tracing Method", *Programming and Computer Software*, **25**:2 (1999), pp. 100–107.

[3] S. M. Abramov, V. A. Vasenin, Ye. Ye. Mamchits, V. A. Roganov, A. F. Slepukhin. "Dynamic parallelization of programs based on parallel graph reduction. A software architecture of new T-system version", *Vysokoproizvoditel'nyye vychisleniya i ikh prilozheniya*, Trudy Vserossiyskoy nauchnoy konferentsii (30 oktyabrya–2 noyabrya 2000 g., g. Chernogolovka), Izd-vo MGU, M., 2000, pp. 261–264 (in Russian).

[4] A. Moskovsky, V. Roganov, S. Abramov. "Parallelism granules aggregation with the T-system", *Proceedings of the 9th international conference on Parallel Computing Technologies*, PaCT'07 (Pereslavl-Zalessky, Russia, September 3–7, 2007), Lecture Notes in Computer Science, vol. **4671**, Springer-Verlag, Berlin–Heidelberg, 2007, pp. 293–302.

[5] V. A. Roganov, A. A. Kuznetsov, G. A. Matveyev, V. I. Osipov. "Methods of adaptation of the OpenTS parallel programming system runtime for the hybrid computing clusters", *Programmnye Sistemy: Teoriya i Prilozheniya*, **4**:4(18) (2013), pp. 17–31 (in Russian), URL: http://psta.psiras.ru/read/psta2013_4_17-31.pdf

[6] A. I. Adamovich. "Fibers as the basis for the implementation of the notion of the T-process for the JVM platform", *Programmnye Sistemy: Teoriya i Prilozheniya*, **6**:4(27) (2015), pp. 177–195 (in Russian), URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf

[7] Eclipse Text Modelling Framework (Xtext), URL: http://www.eclipse.org/Xtext

[8] Eclipse IDE, URL: http://www.eclipse.org

[9] T. Parr. *ANTLR (ANother Tool for Language Recognition)*, URL: http://www.antlr.org

[10] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*, 2nd edition, Safari Books Online, 2008, URL: https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885

[11] L. Bettini, Java−−, URL: https://github.com/LorenzoBettini/javamm

[12] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*, 1st., MIT Press and McGraw-Hill, 1990.

[13] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, A continually updated technical report, University of Virginia, Charlottesville, Virginia, 1991–2007, URL: http://www.cs.virginia.edu/stream/

[14] J. Hammond. *STREAM*, URL: https://github.com/jeffhammond/STREAM/tree/master

[15] *The Go Programming Language Documentation*, URL: https://golang.org/doc/

[16] S. Ajmani. "Program your next server in Go", *Proceedings of Applicative 2016* (New York University, June 1st–2nd, 2016), ACM, New York, USA, URL: https://talks.golang.org/2016/applicative.slide#1

[17] C. Doxsey. *Introducing Go*, O'Reilly Media, 2016, ISBN: 978-1-4919-4195-9, 128 p.

[18] R. L. Bocchino, Jr. , V. S. Adve , S. V. Adve, M. Snir. "Parallel programming must be deterministic by default", *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09 (March 30–31, 2009, Berkeley, California), pp. 4-4.

[19] R. Bocchino, et al. "A type and effect system for deterministic parallel Java", *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09 (October 25–29, 2009, Disney's Contemporary Resort, Orlando, Florida), ACM, New York, NY, USA, 2009, pp. 97–116, ISBN: 978-1-60558-766-0.

[20] A. V. Slesarenko. "Scalan: polytypic library for nested parallelism in Scala", *Keldysh Institute preprints*, 2011, 022, 28 p., URL: http://library.keldysh.ru/preprint.asp?id=2011-22&lg=e

[21] G. E. Blelloch. "Programming parallel algorithms", *Communications of the ACM*, **39**:3 (March 1996), pp. 85–97.

[22] Akka Documentation, URL: http://akka.io/docs/

[23] M. K. Gupta. *Akka Essentials*, Packt Publishing, Birmingham, UK, 2012, ISBN: 978-1-84951-828-4.

[24] C. Hewitt, P. Bishop, R. Steiger. "A universal modular ACTOR formalism for artificial intelligence", *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73 (August 20–23, 1973, Stanford, California, USA), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.

Sample citation of this publication:

Alexei Adamovich. "The Ajl programming language: the automatic dynamic parallelization for the JVM platform", *Program systems: theory and*

applications, 2016, **7**:4(31), pp. 83–117. (*In Russian*).
URL: http://psta.psiras.ru/read/psta2016_4_83-117.pdf