ДЕТЕРМИНИРОВАННЫЕ ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ С МОНОТОННЫМИ ОБЪЕКТАМИ[1]

Анд.В. Климов

Аннотация. Обсуждается параллельная модель вычислений с монотонными объектами, обладающая детерминированностью результатов вычислений. Она может быть определена как расширение функционального языка объектами классов, удовлетворяющих определенным ограничениям. Понятие монотонных объектов и монотонных классов определяется так, чтобы их использование из функционального языка с параллельной операционной семантикой сохраняло детерминированность вычислений, присущую функциональным языкам. Детерминированность языка определяется через два свойства: конфлюэнтность и повторную вычислимость. Приводится пример простого монотонного класса, и обсуждаются свойства и особенности этой модели вычислений.

Введение: мотивировка

Рассмотрим некоторый язык программирования с семантикой, основанной на параллельных вычислениях. Параллельность предполагает недетерминированность порядка вычислений, что приводит в общем случае к недетерминированности результатов вычислений. Зададимся вопросом: *можно ли и как создавать параллельные программы, которые дают однозначный результат, независимый от порядка вычислений?*

Известны следующие крайние подходы к решению этой задачи:

1) языки, у которых все программы имеют *детерминированную* семантику. Это значит, что несмотря на наличии параллелизма в реализации языка, результаты вычислений определены однозначно независимо от порядка вычислений. Таковы все чисто функциональные языки: в композиции функций $f(g(x), h(y))$ два вызова $g(x)$ и $h(y)$ могут вычисляться параллельно в языках со строгой семантикой и все три вызова $f(\_,\_)$, $g(x)$ и $h(y)$ — в языках с нестрогой семантикой. С точки зрения практического параллельного программирования такие языки ограничены. В частности, без специализированного расширения на них нельзя эффективно представлять и обрабатывать произвольные графы, а только деревья;

2) универсальные языки параллельного программирования, на которых можно писать *недетерминированные* программы. Таковы большинство языков. Как правило, понятие параллельных процессов (тредов, threads) представлено в них явно. Чтобы гарантировать, что программа выдает детерминированные результаты, нужно проводить доказательство этого факта специально для данной программы.

Рассмотрим промежуточную модель вычислений, использующую оба подхода. Для этого сконструируем язык программирования из двух подмножеств:

1) все программы на первом подмножестве *детерминированы* по построению (как в подходе 1);
2) программы на втором подмножестве могут быть *недетерминированными* (как в подходе 2).

Отношение подъязыков друг к другу может быть различным: подмножества могут не пересекаться, или второе подмножество может совпадать со всем языком, а первое являться его частью. В качестве недетерминированного подъязыка можно использовать универсальный объектно-ориентированный язык, скажем, Java, а в качестве детерминированного — некоторый чисто функциональный язык, расширив его средствами взаимодействия с объектно-ориентированным языком: ввести тип данных «ссылка на объект некоторого класса» и синтаксис для записи вызова метода в объекте, а также оговорить в семантике, как и когда формируются параллельные процессы.

Методику детерминированного программирования на таком «составном» языке можно представить следующим образом:

- на недетерминированном подмножестве «эксперты» пишут «системные подпрограммы» (классы, если речь идет об объектно-ориентированном языке), для которых авторами проводится доказательство, что при их вызове из программ на первом подмножестве детерминированность сохраняется;
- на детерминированном подмножестве «простые программисты» пишут прикладные программы, вызывающие системные подпрограммы, без каких-либо формальных ограничений. Детерминированность результатов вычисления гарантируется по построению языка и по доказанным свойствам системных подпрограмм.

Классы и объекты, которые по этой методике можно использовать из функционального языка, назовем *монотонными.* Ниже дадим более точное определение.

---

[1] Опубликовано в сб.: Научный сервис в сети Интернет: многоядерный компьютерный мир. 15 лет РФФИ: Труды Всероссийской научной конференции (24-29 сентября 2007 г., г. Новороссийск). — М.: Изд-во МГУ, 2007. С. 212-217.

Язык с монотонными объектами предоставляет более выразительные средства, чем чисто функциональные языки, при сохранении их положительных свойств. Такого языка достаточно, чтобы эффективно строить и обрабатывать более сложные, чем деревья, структуры данных, произвольные графы. Покажем на примере, как с помощью монотонных объектов можно передавать информацию между параллельными вызовами функций, что невозможно на чисто функциональном языке.

Пример

Рассмотрим пример, используя широко распространенные языки: объектно-ориентированный язык Java и функциональный язык SML [4], ограничив его до чисто функционального подмножества и расширив следующим образом.

Добавим в SML оператор вызова конструктора `new` *ИмяКласса*`(`*АргументыКонструктора*`)`, вырабатывающий значение типа *ИмяКласса* — ссылку на новый объект, и оператор вызова метода *x*`.`*ИмяМетода*`(`*Аргументы*`)`, где *x* — переменная типа *ИмяКласса*. Некоторым естественным способом отобразим базовые типы языков SML и Java друг в друга. Для примера нам будет достаточно типа int, общего для SML и Java, и отображения отсутствующего результата Java-метода `void` в тип `unit` языка SML, единственное значение которого записывается как `()`.

Сохраним присущую SML строгую семантику: функции вызываются после полного вычисления аргументов. Введем в SML параллелизм, считая, что все независимые вызовы функций вычисляются параллельно, т.е. в композиции $f(g(x), h(y))$ для вычисления $g(x)$ и $h(y)$ создаются два новых процесса (нити, threads), а вычисление $f(\_,\_)$ начнется только после их завершения. (Возможны, и даже желательны реализации, которые заботятся о том, чтобы не вычислять лишнего, приближая семантику к ленивым вычислениям, сохраняя параллелизм, но для определения семантики это неважно.)

Вызываемые из SML Java-методы могут вырабатывать исключения (exceptions). Однако, для простоты не будем разрешать их перехват в детерминированном подмножестве языка SML. В определении детерминированности не будем учитывать случаи, когда один из вариантов вычислений выдает исключение. Нам важна лишь эквивалентность нормальных результатов вычислений.

Пример монотонного класса

Опишем на языке Java класс `IntVar`, напоминающий переменную с однократным присваиванием значения типа `int`, со следующими операциями:

- вызов конструктора `new IntVar()` создает «пустой» объект с неопределенным значением;
- метод `void x.set(int i)` сохраняет значение `i` в объекте `x`. Если в том же объекте второй раз вызывается метод `x.set(j)` с *неравным* аргументом, то вырабатывается исключение (`RuntimeException`);
- метод `int x.get()` возвращает значение `i`, если оно уже было записано в объект `x` вызовом `x.set(i)`, или переходит в состояние *ожидания*, пока другой процесс не вызовет `x.set(i)`.

```
public class IntVar {

    boolean empty = true;
    int value;

    public synchronized int get()
        throws java.lang.InterruptedException
    {
        if (empty) wait();
        return value;
    }

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

Отметим, что вызовы `x.get()` и `x.set(i)` можно исполнять в любом порядке и многократно. Их результат и побочный эффект будут эквивалентными. Можно доказать, что эти свойства — независимость от порядка вычислений и возможность повторно вычислять копию любого вызова функции — сохраняется и в функциональном языке, использующем объекты этого класса.

Пример использования монотонного класса

Ниже приведено определение на языке SML трех функции, использующих класс `IntVar`:

- `intlist n` создает список длины `n` из ссылок на новые объекты типа `IntVar`, которые находятся в состоянии «неопределено» (`empty==true`). В тексте функции `intlist` используются константа `[]` — пустой список и операция `::` — добавление элемента впереди списка;
- `setfibs xs` заполняет объекты типа `IntVar`, ссылки на которые лежат в списке `xs`, числами Фибоначчи в *обратном* порядке: в первый объект записывается `fib(n)`, где `n` — длина `xs`, в последние два объекта — `fib(2) = fib(1) = 1`. Функция рекурсивно порождает процессы, вычисляющие выражения `x.set(y.get() + z.get())`, где `x`, `y`, `z` — последовательные три элемента списка `xs`. Эти

процессы порождаются слева направо для каждого x из списка xs кроме двух последних, а вычисляются они справа налево в процессе готовности значений, извлекаемых вызовами y.get() и z.get(). В последние два объекта вызовами x.set(1) записывается 1 согласно второму предложению функции setfibs. Пояснение по языку SML: в левых частях предложений используются образцы (x :: y :: z :: xs) и (x :: xs); первый выделяет три левых элемента x, y, z из списка, если его длина не меньше 3, оставляя «хвост» в xs; второй выделяет головной элемент x, когда он есть и первый образец неприменим, то есть когда длина списка равна 1 или 2. В правых частях предложений выражение (*A*; *B*) означает «вычислить *A* и забыть его значение, вычислить *B* и выдать его значение как результат всего выражения»; в нашей семантике *A* и *B* вычисляются параллельно;

- fib n вычисляет n-ое число Фибоначчи искусственным способом, интересным лишь для демонстрации использования функций intlist и setfibs: к списку ссылок на пустые объекты list, выданному функцией intlist, применяется функция setfibs и *параллельно* запрашивается значение из головы списка: (hd list).get(), где hd (от слова «head») — функция из стандартной библиотеки, выдающая первый элемент списка.

```
fun intlist 0 = []
  | intlist n = new IntVar() :: intlist (n-1)

fun setfibs (x :: y :: z :: xs) = (x.set(y.get() + z.get());
                                   setfibs (y :: z :: xs))
  | setfibs (x :: xs) = (x.set(1);
                         setfibs xs)
  | setfibs [] = ()

fun fib n = let val list = intlist n
            in  setfibs list;
                (hd list).get()
            end
```

На рис. 1 изображено одно из возможных промежуточных состояний вычисления fib(5). Объекты изображены овалами, процессы прямоугольниками, ссылки стрелками. Последние в списке три объекта уже получили значения fib(3) = 2, fib(2) = 1 и fib(1) = 1; вычислившие их процессы завершились. Активны два процесса, вычисляющие подвыражения вида x.set(y.get() + z.get()). Значения x, y и z являются ссылками, изображенными черными кружками на конце стрелок, ведущих к соответствующему объекту. На следующих шагах каждый из этих процессов создаст новые параллельные процессы для вычисления y.get() и z.get(), а их результатов будут ожидать процессы x.set(_ + _). Процесс hd[●,●,●,●,●].get() также активен. На следующем шаге он преобразуется в ●.get() и после вызова метода get() встанет на ожидание вызова ●.set(5) в первом объекте списка.

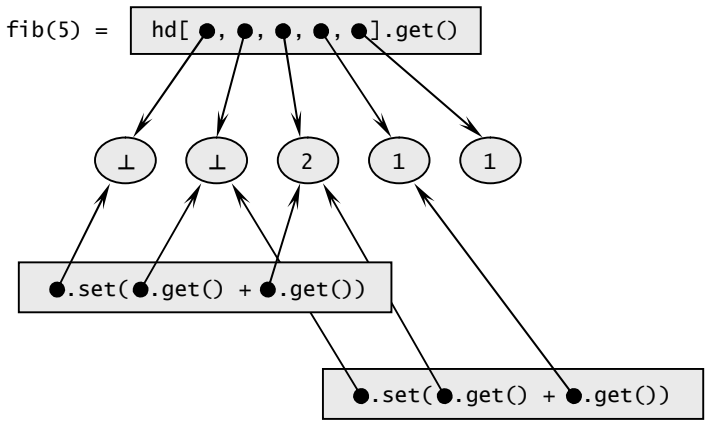

Рис. 1. Некоторое промежуточное состояние вычисления fib(5).

Перейдем к более строгому определению монотонности, уточнив сначала понятие эквивалентности значений со ссылками и использующие его понятия конфлюэнтности и повторной вычислимости.

Эквивалентность значений со ссылками

Введение в функционального язык ссылок на объекты в качестве значений нарушает его *референциальную прозрачность*, которая операционно означает, что повторное вычисление подвыражения с теми же значениями дает *равный* результат, т.е. вычисление выражения f(x) = f(x) всегда дает true, когда оно завершается. Это свойство не выполняется для конструкторов объектов: выражение new C(x) = new C(x) всегда вырабатывает false. Таким образом, если вычисление f(x) завершается, то f(x) = f(x), когда значение f(x) не содержит новых ссылок, сгенерированных в процессе вычисления f(x), и f(x) ≠ f(x) в противном случае. Тем не менее, при вычислениях с

монотонными объектами имеет место более слабая *эквивалентность*, определенная следующим образом.

Определение (эквивалентность). Два результата вычисления *x* и *y* некоторых выражений *X* и *Y* являются *эквивалентными*, если для любой функции `f`, определенной на данном языке, результаты вычисления `f(`*x*`)` и `f(`*y*`)` равны в том случае, если они выдаются за конечное время и не содержат в себе ссылок на объекты.

Другими словами, значения, содержащие ссылки на объекты, эквивалентны, если на языке программирования нельзя обнаружить в них разницу. Эквивалентные объекты содержат в себе одинаковую информацию, которую можно из них извлечь. С точки зрения «внешнего наблюдателя» они неотличимы. Если эквивалентные объекты используются для представления графов, то эти графы равны.

Понятия монотонного класса и монотонного объекта

Представление о детерминированной семантике языка можно разложить на два операционных понятия. Первое — независимость *от порядка* параллельных вычислений: вычисления в разном порядке дают *эквивалентные* результаты. Это свойство также называют *конфлюэнтностью* или *свойством Черча-Россера.*

Второе менее очевидно: мы хотим, чтобы тот факт, что вызов функции имеет некоторое значение, был независим от того, сколько раз он вычислялся и в каком контексте. Это — независимость *от кратности* вычислений вызова функции. Операционно это формулируется так: если в любой момент вычислений откопировать некоторый готовый к вычислению вызов функции *f*(*x*), затем через некоторое количество шагов запустить вычисление копии как параллельный процесс, то результат вычисления копии будет *эквивалентным* исходному. Назовем это свойство *повторной вычислимостью*. Отметим, что оно не связано с параллелизмом, но его добавление к конфлюэнтности заметно упрощает построение мира детерминированных параллельных вычислений. В теории, благодаря ему можно дать естественную *детонационную семантику* языку (так, чтобы каждое подвыражение языка имело значение, *денотат*). На практике это свойство не менее важно: благодаря ему можно повысить *надежность* за счет перевычисления после сбоев и отказов оборудования.

Определение (монотонность). Назовем класс, определенный на объектно-ориентированном языке, и объекты, принадлежащие этому классу, *монотонными*, если при его использовании из функционального языка по описанной выше схеме для всех программ на функциональном языке сохраняются два свойства:

1) конфлюэнтность (свойство Черча-Россера);

2) повторная вычислимость вызовов функций.

Множество определений монотонных классов на универсальном объектно-ориентированном языке алгоритмически неразрешимо и неперечислимо. Поэтому конструирование монотонных классов и доказательство их монотонности — дело человека.

Непрозрачность объектов для ссылок

Вернемся к примеру монотонного класса `IntVar`. Зададимся вопросом: можно ли по этому же шаблону определить монотонный класс `ObjectVar` для сохранения в нем произвольных объектов. Формально заменим `int` на `Object` и выпишем отличающиеся строки и еще одну, на которую нам надо обратить внимание:

```
public class ObjectVar {
    ...
    Object value;

    public synchronized Object get()
    ...
    public synchronized void set(Object x) {
        ...
        else if (value != x) throw new RuntimeException();
    }
}
```

Определим на SML функцию `setnew x`, которая присваивает в объект `x` класса `ObjectVar` новый объект типа `IntVar`. Затем в функции `setnew2` вызовем `setnew x` два раза с одним и тем же объектом:

```
fun setnew x  = x.set(new IntVar())
fun setnew2() = let val x = new ObjectVar() in setnew x; setnew x end
```

Если бы класс `ObjectVar` был монотонным, вызов `setnew2()` вычислился бы без проблем. Однако, при втором вызове `setnew x` методу `set` в объекте `x` будет передана ссылка на новый объект класса `IntVar` и после проверки `value != x` будет выработано исключение.

Чтобы разрешить эту проблему, придется пожертвовать сохранением ссылок на объекты при их записи и чтении в/из объектов, и в методе `get` выдавать ссылку на клон сохраненного объекта или на новый объект-посредник, который будет представлять тот же объект другой ссылкой. Тогда в методе `set` вместо проверки на равенство по ссылке достаточно будет проверять на эквивалентность объектов по их

содержимому. Это сложнее в реализации, но нет другого способа построения монотонных объектов, позволяющих строить циклические структуры данных и произвольные графы.

Такое свойство назовем «непрозрачностью объектов для ссылок». Аналогично можно показать, что ему удовлетворяют все монотонные объекты. С другой стороны, сохранение ссылок на объекты, передаваемые через аргументы конструкторов, не нарушает монотонности.

Заключение: история, проблемы, близкие и будущие работы

Разработка модели вычислений с монотонными объектами была начата в 80-е годы [1, 2, 3]. Тогда было обнаружено, что ее реализация в полном объеме сопряжена с заметной потерей эффективности из-за необходимости подавлять «взрывной параллелизм» с помощью организации вычислений по запросам (demand-driven computation), которые требуют обратных ссылок от объектов к использующим их процессам. Основной путь повышения эффективности — анализ и специализация программ с последующим выделением частных случаев, которые могут быть реализованы эффективными шаблонами программирования. Оптимизм основан на том, что программы с монотонными классами так же хорошо поддаются специализации [3], как и чисто функциональные, — по крайней мере, гораздо лучше, чем с классами общего вида. С тех пор достигнут заметный прогресс в методах специализации программ как на функциональных, так и на объектно-ориентированных языках. Мы планируем перенести их на языки с монотонными объектами.

Виды данных, напоминающие монотонные объекты, распространены в программирования. Самый яркий пример — логические переменные в Прологе и в более поздних работах по интеграции функциональных и логических языков на основе narrowing-driven computation [5, 6]. Однако, в этих и других известных нам моделях вычислений, использующих «недоопределенные» данные, в явном виде ссылки на объекты не вводятся. Здесь объекты и ссылки — лишь «колесики реализации», а на уровне семантики языка эти понятия предстают в виде частично определенных значений, монотонно доопределяемых в процессе вычислений. Введение же явных ссылок на монотонные объекты дает возможность различать обычные значения и потенциально недоопределенные, предоставляет средства инкапсуляции данных и механизм интеграции функциональных и объектно-ориентированных языков. За это приходится платить лишь отказом от референциальной прозрачности в ее точном виде, но с сохранением в некоторой ослабленной формулировке на основе эквивалентности значений вместо их равенства.

Основные темы для дальнейшего исследования: эффективная реализация как на последовательных, так и параллельных компьютерах; разработка библиотеки монотонных объектов и образцов прикладных программ; развитие и реализация методов специализации и верификации программ с монотонными объектами.

Работа выполняется при поддержке проекта РФФИ № 06-01-00574 и проекта Роснауки № 2007-4-1.4-18-02-64.

ЛИТЕРАТУРА:

1. А.В. Климов "Применение концепции смешанных вычислений для конструирования объектно-ориентированного функционального языка программирования" // Семиотические аспекты формализации интеллектуальной деятельности. Школа-семинар "Телави-83". Тезисы докладов и сообщений, М.: ВИНИТИ, 1983, с.67-70
2. А.В. Климов "Объектно-функциональная модель вычислений" // Семиотические аспекты формализации интеллектуальной деятельности. Школа-семинар "Боржоми-88". Тезисы докладов и сообщений, М.: ВИНИТИ, 1988, с.49-53
3. A.V. Klimov "Dynamic specialization in extended functional language with monotone objects" // Proc. 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91 (New Haven, Connecticut, United States, June 17-19, 1991), ACM Press, 1991, pp.199-210
4. R. Milner, M. Tofte, R. Harper, D. MacQueen "The Definition of Standard ML - Revised", MIT Press, 1997
5. U.S. Reddy "Narrowing as the operational semantics of functional languages" // Proc. IEEE Internat. Symposium on Logic Programming, Boston, 1985, pp.138-151
6. M. Alpuente, M. Hanus, S. Lucas, G. Vidal "Specialization of Functional Logic Programs Based on Needed Narrowing" // Theory and Practice of Logic Programming 5(3), 2005, pp.273-303