УДК 004.432 + 004.422

ИСПОЛЬЗОВАНИЕ ОБОБЩЁННЫХ КОНЦЕПЦИЙ ИТЕРАТОРОВ И ФУНКЦИОНАЛЬНЫХ ОБЪЕКТОВ ПРИ РЕШЕНИИ ЗАДАЧ МАТЕМАТИЧЕСКОГО МОДЕЛИРОВАНИЯ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ С ПОДДЕРЖКОЙ ПАРАМЕТРИЧЕСКОГО ПОЛИМОРФИЗМА ПОДТИПОВ

© 2018 Д.Е. Яблоков

Самарский национальный исследовательский университет имени академика С.П. Королёва

Статья поступила в редакцию 12.12.2018

В статье рассматривается альтернативный подход к научному программированию, обеспечивающий необходимый уровень гибкости и структурной целостности кода при разработке обобщенных алгоритмов и структур данных. При решении задач математического моделирования, связанных с научными исследованиями, использование чисто объектно-ориентированных языков, таких как C# и Java, происходит достаточно редко. Исключая вопросы быстродействия, это связано с недостаточно развитыми средствами итерирования и доступа к элементам коллекций, а также отсутствием информации об истинном типе в универсальных объявлениях во время компиляции. Для устранения этих недостатков в статье предлагается использование обобщённых концепций, позволяющих специфицировать требования к аргументам алгоритмов. Возможность контроля предусловий выполнения, основанная на синтаксических и семантических особенностях, задаёт требуемую стратегию обработки данных. Понятие концепции широко известно и применяется в технологиях программирования как спецификация требований к параметрам типов в предикативной форме. Эти требования устанавливают правила для ожидаемой семантики и синтаксической общности языковых конструкций и дают понимание того, что должны представлять собой связанные с ними типы данных, литералы, допустимые операции и выражения. Статья предназначена для постоянно практикующих специалистов, использующих C++, C# или Java в научной работе и будет интересна с точки зрения концептуального объединения многих, уже существующих, понятий, методов и технологий обработки данных.

Ключевые слова: Обобщённый алгоритм, обобщённая концепция, нотация диапазона, способ обхода диапазона, концепция итератора, концепция функционального объекта, объектно-ориентированный язык, параметрический полиморфизм подтипов.

1. ВВЕДЕНИЕ

Согласно [1] научное программирование – это совокупность необходимых и доступных для использования методов, теоретических и технологических приемов для решения задач, возникающих в исследовательской деятельности. К такому теоретико-технологическому направлению можно отнести рассматриваемый в статье подход, цель которого поиск наиболее общих элементов программного решения подходящим образом отвечающих стратегии объектной и процедурной декомпозиции.

Как и для других задач математического моделирования, процесс формирования решения вычислительных задач связан с определенной стадийностью. На первом этапе необходимо провести постановку задачи, сформулировав её с точки зрения особенностей выбранной предметной области. Далее следует математическая постановка, когда проблема транслируется на математический язык, средствами которого модель могла бы корректно описывать исследуемые процессы. Затем, оперируя методами абстрактной математики, решение описывается в общем виде, без использования конечных чисел, с применением формул, функций и абстрактных величин. После этого, с помощью алгоритмизации и программирования, решение задачи пытаются свести к выполнению конечного количества арифметических операций и упорядочить производимые действия в виде точного, воспроизводимого метода с программной реализацией на языке высокого уровня. Этапам алгоритмизации и программирования в статье уделяется особое внимание, потому что именно на этих стадиях формируется понятийный аппарат и инструментарий, с помощью которого будут производиться расчёты и анализ результатов. Акцент специально не делается на решениях и реализациях для конкретных задач. Основная цель статьи – дать краткий обзор некоторых категорий идей, которые соответствуют фундаментальным понятиям процедурного, объектно-ориентированного и обобщённого программирования, чтобы пояснить суть предлагаемого направления.

Яблоков Денис Евгеньевич, ведущий инженер Межвузовского научно-исследовательского центра по теоретическому материаловедению. E-mail: dyablokov@gmail.com

2. ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ ПОДТИПОВ

Термин полиморфизм буквально означает условие для возможности существования нескольких форм или видов чего-либо. В программировании – это способность связывать различные специфические варианты поведения с помощью единого методологического подхода.

В объектно-ориентированной парадигме [2] полиморфизм реализуется посредством виртуальных функций и наследования, что называется динамическим полиморфизмом или полиморфизмом подтипов. Требования к поведению могут быть записаны в виде интерфейса, наследуя которому конкретные классы или подтипы определяют реализации присутствующих в сигнатуре базовой абстракции методов. Хорошим примером подобного подхода является аддитивная группа как множество элементов с ассоциативной операцией сложения, операцией аддитивной инверсии и нейтральным элементом.

```
interface IAdditiveGroup
{
   void Plus(IAdditiveGroup that);
   IAdditiveGroup Negate();
   IAdditiveGroup Zero{get;}
};
```

Если использовать этот интерфейс как базовый контракт, то появляется возможность реализации универсальных операций. Например, функция суммирования может работать с любыми массивами, типами элементов которых являются классы производные от `IAdditiveGroup` и, соответственно, реализующие все методы и свойства, объявленные у предка.

```
IAdditiveGroup Sum(IList<AdditiveGroup> grps,
   IAdditiveGroup init)
{
   var res = init;
   foreach(var grp in grps)
      res.Plus(grp);
   return res;
}
```

Примерами таких классов могут быть реализации различных чисел или же объектов-компоновщиков структурирующих другие объекты в иерархии по типу часть-целое, что позволяет единообразно трактовать поведение, как индивидуальных, так и составных объектов. Согласно [3] объектно-ориентированное программирование часто называют программированием с организованной в виде таксономии [4] семантикой и данными, что создает условия для эффективного структурирования концептуальных границ программного решения. Полиморфизм в этом случае достигается за счет представления функциональных особенностей в терминах базового контракта, так, чтобы любой подтип был взаимозаменяем и, во время выполнения вызывалась соответствующая реализация конкретного объекта-потомка.

Полиморфизм в обобщённом программировании реализуется за счёт параметризации типов, присутствующих в сигнатурах классов и функций, что называется статическим или параметрическим полиморфизмом [5]. Вместо определения фактора зависимости от общего поведения свойственного иерархиям с корнем в виде интерфейса или абстрактного базового класса во главу угла ставится общность подразумевающая поддержку общего синтаксиса. Объекты со схожим интерфейсом могут определяться независимо друг от друга, а параметры типов формируют семейства задающие требования к возможным кандидатам на роль конкретной специализации типа аргумента.

```
template<template<typename> class AG,
   size_t N, typename T >
T Sum(array<AG<T>, N> grps,
   T const & init)
{
   auto res = init;
   for(auto & grp : grps)
      res += grp;
   return res;
}
```

Из приведенного на C++ примера видно, что наличие базового контракта `IAdditiveGroup` для спецификации поведения алгоритма уже не нужно, хотя, для определения неявно выраженной концепции, продиктованного ожиданиями к синтаксису и семантике аргумента, акроним AG присутствует в качестве имени параметра типа. Любой тип удовлетворяющий требованиям, обозначенным в теле алгоритма для AG должен быть регулярным [6], а также поддерживать операцию суммирующего присваивания `operator += (T const &)` и операцию приведения типа `operator T() const`. Полиморфизм при этом проявляется за счет специализации параметров типов конкретными классами, интерфейсы которых не определены заранее и могут охватывать только часть синтаксических и семантических характеристик, необходимых для корректной работы алгоритма.

Основное отличие механизмов обобщения использующихся в C++ и чисто объектно-ориентированных языках, таких как C# или Java, состоит в том, что шаблоны C++ не требуют и, на данный момент не поддерживают, каких-либо ограничений при параметризации типов [7, 8]. В C++ проверка конкретных типов в универсальных объявлениях осуществляется только в момент их привязки к параметрам типов. В C# и Java присутствует возможность наложения ограничений на параметры типов, в виде характеристик которыми должен обладать конкретный тип, что позволяет проверять это соответствие независимо от момента использования. На этом

свойстве, использующем особенности дынных языков, и называющимся параметрическим полиморфизмом подтипов, основано применение обобщённых концепций.

```
R Accumulate<IFI, T, R>(IFI first, IFI last,
   R init, Func<R, T, R> oper)
   where IFI :
ForwardIterator<IFI>,
IInputValue<T>
{
   if(!first.CompatibleTo(last))
      /* throw an exception */
   var _res = init;
   var _first = first.New();
   while(_first != last)
   {
      _res = oper(_res, _first.Value);
      ++_first;
   }
   return _res;
}
```

Из приведенного кода видно, что обобщённый алгоритм выполняет некоторую бинарную операцию oper для всех значений диапазона `[first, last)`. Обозначающие границы объекты, должны поддерживать проверку на совместимость относительно идентичности представляемого набора данных `first.CompatibleTo(last)`, создание копии с сохранением состояния `first.New()`, сравнение на эквивалентность `_first != last`, продвижение на одну позицию вперед `++_first`. Также, кроме функциональных особенностей, обеспечивающих общую семантику обхода, должна присутствовать возможность доступа к значению элемента в текущей позиции в режиме чтения `_first.Value`. Являясь именованным набором ограничений для одного или нескольких параметров типов аргументов, обобщённые концепции `ForwardIterator<IFI>` и `IInputValue<T>`, совместно с бинарной операцией `Func<R, T, R>`, позволяют задавать необходимую стратегию поведения. Но, они определяют только тот набор абстрактных требований, который важен для работы данного алгоритма или ему подобных. При более развитой семантике, когда в теле алгоритма используются выражения, требующие поддержки обхода в обратном направлении, произвольного доступа, записи значения элемента, охватываться должен больший объем функциональных особенностей, обеспечивающих правильный и ожидаемый, в соответствии требованиями, режим работы.

3. ОБОБЩЁННЫЕ КОНЦЕПЦИИ И МОДЕЛИРОВАНИЕ

Концепция – это основанная на целостных и систематизированных представлениях совокупность взглядов, позволяющая выражать определённый способ понимания [9, 10] или трактовки чего-либо. Она задаёт стратегию действий направленную на поиск путей решения выбранной задачи в рамках имеющегося понятийного аппарата, а также средств и технологий, допустимых для получения этого решения. С точки зрения программирования можно сказать что концепция – это спецификация набора требований, дающих представление о свойствах и ограничениях, которыми должен обладать тип данных, моделирующий эту концепцию $C \in \{r_1^T, r_2^T, \dots, r_n^T\}$, где r_i^T – элемент множества абстрактных требований которым должен соответствовать тип данных. Концепцию также можно рассматривать как семейство или набор типов с общими для всех его элементов характеристиками. Например, если тип T является моделью концепции C, то это означает, что $T \in \{t_1^C, t_2^C, \dots, t_n^C\}$, где t_i^C – элемент множества типов, которое представляет C, а n – мощность этого множества. Кроме того, концепцию C можно представить в виде списка допустимых алгоритмов. В соответствии с этим определением значение и суть концепции заключается в её использовании во множестве алгоритмов, а именно: $C \in \{a_1^T, a_2^T, \dots, a_n^T\}$, где a_i^T – элемент множества доступных для использования типа T алгоритмов, а n – его мощность.

Новые концепции и их свойства открываются и описываются путём изучения того, каким требованиям должны соответствовать параметры обобщённых алгоритмов. Полезность любого алгоритма повышается, если его разработка ведётся не в терминах конкретных типов, а в терминах требований к ним, то есть концепций, выражающих как синтаксические, так и семантические свойства. Для типов, моделирующих концепцию, должна присутствовать возможность реализации свойств и ограничений с учётом особенностей выбранного языка. Используемые понятия должны выражать общие направления, а не просто артефакты инструмента программирования. Рудиментарная система воззрений на концепции, как ещё на один предметно-ориентированный язык [11], может привести к частично сформированным идеям. Вместо того чтобы сосредотачиваться на требованиях к конкретным функциям или интерфейсам, концепции воздействуют на целевую предметную область в более широком смысле и представляют полностью сформированные абстракции.

Рассматриваемые в статье обобщённые концепции итераторов и функциональных объектов – это взаимодополняющие наборы ограничений. Они определяют, с помощью параметрического полиморфизма подтипов, соответствие используемых синтаксических конструкций и, на их основе, дают гарантии семантических свойств типов и допустимых операций. Источником идей для создания такого подхода послужила стандартная библио-

тека шаблонов STL [12] языка С++, стереотипы и идиомы которой имеют всеобщий характер и могут быть применимы как основополагающие принципы при разработке обобщённых алгоритмов.

4. ДИАПАЗОНЫ И СПОСОБЫ ОБХОДА

В общем смысле понятие диапазон [13] служит своего рода абстракцией для чего-либо, что обладает признаками начала и окончания и где можно двигаться в определённом направлении с помощью средств, способных выполнять операцию перехода на следующую позицию в зависимости от установленных правил. Существует несколько нотаций диапазонов [6], отличающихся способами задания ограничивающих условий и семантикой средств обхода. Первая из них – это нотация, представляющая нестрогий или слабый диапазон у которого для определения верхней границы используется какой-либо признак (рис. 1).

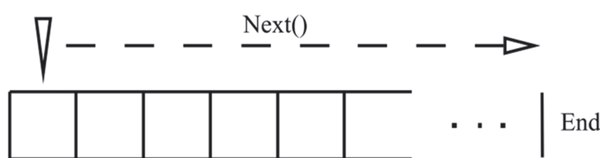

Рис. 1. Нотация нестрогого диапазона

Способом обхода нестрогого диапазона может служить следующая форма кода, когда количество итераций заранее не определено, а завершение процесса обработки, т.е. выход из тела цикла, возможно, если признак окончания перейдёт в сигнальное состояние.

```
while(!End)
{
   /* code */
   Next();
}
```

Второй вид диапазона – это счётный диапазон. Он является развитием понятия нестрогого диапазона с возможностью указания размера, но в отличие от предыдущей нотации гарантирует, что в пределах обрабатываемого интервала не будет циклических элементов и не возникнет замкнутых путей перебора (рис. 2).

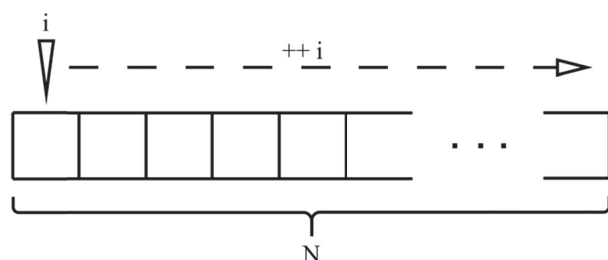

Рис. 2. Нотация счётного диапазона

Код для обхода счётного диапазона, как и в предыдущем примере, может быть представлен в виде циклического оператора, но критерием завершения процесса служит оценка состояния индекса текущей позиции. В пределах обрабатываемого интервала он не должен быть равным или превышать установленный размер.

```
for(/* ... */; i < N; ++i)
{
   /* code */
}
```

Третья нотация – это нотация ограниченного диапазона, идеологически построенная на принципах первых двух видов. Как и в нотации для нестрогого диапазона количество итераций заранее не определено, но, как и в случае счётного диапазона возможно получение размера с помощью частичного вычитания (рис. 3).

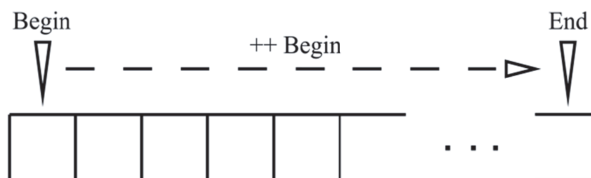

Рис. 3. Нотация ограниченного диапазона

Для этого необходимо, чтобы объекты, обозначающие границы диапазона, соответствовали некоторому типу, поддерживающему произвольный доступ к любой из валидных позиций и операцию вычитания, при которой результат выражал бы расстояние между операндами. Форма кода для организации процесса обхода ограниченного диапазона должна быть такой, что признаком, индицирующим необходимость выхода из тела цикла, является факт обнаружения равенства ограничивающих объектов.

```
for(/* ... */; Begin != End; ++Begin)
{
   /* code */
}
```

Существует четыре разновидности линейных обходов диапазонов, сочетающих в себе все возможные варианты, а именно: однопроходный прямой, многопроходный прямой, двунаправленный (многопроходный прямой и обратный), с произвольным доступом. Остальные способы: однопроходный обратный и многопроходный обратный легко могут быть выражены через предыдущие варианты с незначительными модификациями.

Для некоторых алгоритмов, например, копирования или трансформации [14, 15], достаточна поддержка однопроходной семантики, когда операция перемещения на следующую позицию может осуществляться только единожды для текущего положения, а обращение к возможным копиям, указывающим на предыдущие позиции, недопустимо (рис. 4).

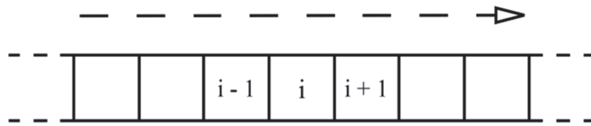

Рис. 4. Нотация однопроходного прямого обхода

Практически это означает, что согласно нотации достаточным является только однократный проход в прямом направлении, то есть допускается совершить только один переход от позиции i−1 к позиции i, а затем от i к i+1.

Алгоритмы, производящие сравнение двух диапазонов или осуществляющие поиск вхождения поддиапазона [14, 15] требуют от своих аргументов поддержки многопроходного прямого обхода, позволяющего обращение к копиям, хранящим внутреннее состояние (рис. 5).

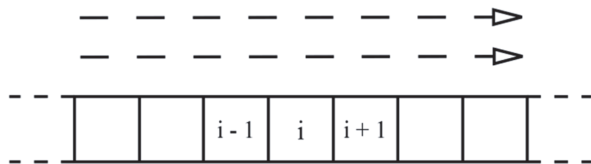

Рис. 5. Нотация многопроходного прямого обхода

В случае многопроходной нотации, остановившись на каком-либо элементе можно сохранить текущее состояние с помощью создания копии, продолжить обход, а затем, вернувшись к сохранённой позиции пройти этот участок повторно.

Для алгоритмов, предназначенных для реверсивного копирования или поиска последнего вхождения поддиапазона [14, 15], необходимым требованием является наличие способности перемещения, в заданных пределах, как в прямом, так и в обратном направлении (рис. 6).

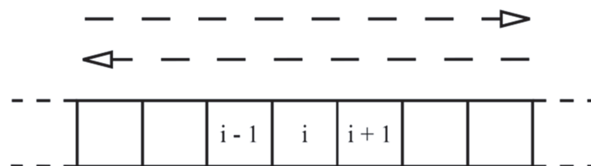

Рис. 6. Нотация двунаправленного обхода

Серия алгоритмов, предназначенных для сортировки, поиска или случайного переупорядочения [14, 15], требует возможности произвольного доступа к любой позиции обрабатываемого диапазона (рис. 7).

Обход диапазона – это одна из наиболее распространённых операций, которые приходится совершать над наборами данных. При подготовке решения вычислительных задач, где будет присутствовать поэлементная обработка линейных последовательностей, предварительная оценка вида диапазона и способа его обхода является важным шагом при проектировании алгоритмов. Именно на этом этапе специфицируются требования к абстракциям, которые в дальнейшем будут представлять данные.

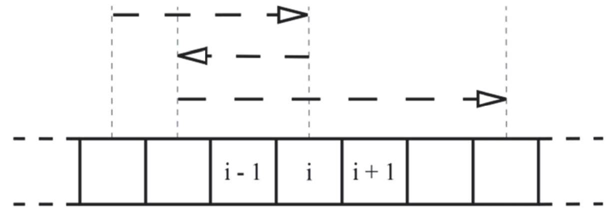

Рис. 7. Нотация обхода с произвольным доступом

5. ОБОБЩЁННЫЕ КОНЦЕПЦИИ ИТЕРАТОРОВ

Обобщённые концепции итераторов важны при проектировании компонентов обработки и анализа данных, так как абстракции, построенные на базе этих понятий, могут являться интерфейсами между обобщёнными алгоритмами и структурами хранения данных. Они также имеют большое значение, поскольку являются обобщением указателей, т.е. объектов, которые указывают на другие объекты и могут использоваться как основа для создания компонентов, осуществляющих проход по диапазону. Если модель концепции итератора указывает на какую-либо позицию диапазона, то после применения операции продвижения она будет указывать позицию, являющуюся следующей или предыдущей, в зависимости от направления обхода. Взаимосвязь концепций итераторов со свойствами указателей не совсем однозначна. Например, указатели в языке C++ имеют очень развитую семантику и к ним могут быть применимы как операции адресной арифметики, так и операции разыменования, получения адреса и сравнения [16]. Обобщённые алгоритмы в большинстве случаев используют лишь малое подмножество свойств указателей в зависимости от той функциональности, на которую они ориентированы. Таким образом, существует несколько различных способов обобщения семантики указателей, при этом каждый из способов является отдельной обобщённой концепцией.

Обобщённая концепция однопроходного прямого итератора подразумевает, что её модели будут поддерживать только однопроходный способ обхода. Это автоматически ограничивает применение только для алгоритмов с линейной сложностью, работающих единовременно только с одной текущей позицией.

```
abstract class TraversalIterator<I> : Concept<I>
{
   protected abstract void Advance();
   public abstract TraversalIterator<I>
      New(TraversalIteratorTag tag);
   public static TraversalIterator<I>
      operator ++ (TraversalIterator<I> iterator)
   {
      iterator.Advance();
      return iterator;
   }
};
```

Обобщённая концепция многопроходного прямого итератора, как развитие концепции однопроходного итератора, в дополнение должна поддерживать операции отношения по критерию эквивалентности и определения идентичности по критерию подобия, чтобы исключить возможность сравнения несовместимых итераторов. Она описывает синтаксические и семантические свойства типов, моделирующих многопроходный прямой способ обхода нестрогих, счётных и ограниченных диапазонов.

```
abstract class ForwardIterator<I> :
   TraversalIterator<I>
{
   protected abstract bool EqualTo(I model);
   public abstract bool
      CompatibleTo(IForwardIterator iter);
   public abstract ForwardIterator<I>
      New(ForwardIteratorTag tag);
   public sealed override TraversalIterator<I>
      New(TraversalIteratorTag tag) =>
         New(new ForwardIteratorTag());
   public override bool Equals(object obj)
   {
      if(obj is null)
         return false;
      if(obj is I model)
         return EqualTo(model);
      if(obj is ForwardIterator<I> iterator)
         return EqualTo(iterator.Model);
      return false;
   }
   public override int GetHashCode() =>
      Model.GetHashCode();

   /* ++, == and != operators */
};
```

Обобщённая концепция двунаправленного итератора обладает всеми качествами многопроходного прямого итератора с дополнительной возможностью единичного сдвига в обратном направлении.

```
abstract class BidirectionalIterator<I> :
   ForwardIterator<I>
{
   protected abstract void Retreat();
   public abstract BidirectionalIterator<I>
      New(BidirectionalIteratorTag tag);
   public sealed override ForwardIterator<I>
      New(ForwardIteratorTag tag) =>
         New(new BidirectionalIteratorTag());

   /* ++ and -- operators */
};
```

Концепция итератора произвольного доступа – это развивающая концепция для двунаправленного итератора. Наряду со свойствами и функциональностью развиваемой концепции она предоставляет дополнительные возможности. Это произвольное смещение в обоих направлениях, получение расстояния в терминах типа данных, представляющего число элементов между итераторами, указывающими на разные позиции одного диапазона, а также определение упорядоченности за счёт операции сравнения двух итераторов по критерию меньше.

```
abstract class RandomAccessIterator<I, D> :
   BidirectionalIterator<I>
{
   protected abstract void Advance(D offset);
   protected abstract void Retreat(D offset);
   protected abstract bool LessThan(I model);
   protected abstract bool GreaterThan(I model);
   protected abstract D Distance(I model);
   public abstract RandomAccessIterator<I, D>
      New(RandomAccessIteratorTag tag);
   public sealed override BidirectionalIterator<I>
      New(BidirectionalIteratorTag tag) =>
         New(new RandomAccessIteratorTag());

   /* ++, --, +, -, <, >, <= and >= operators */
};
```

Ещё одна особенность обобщённых концепций итераторов – это способность доступа к элементам в режиме чтения или записи. Поддержка читаемости [14] для моделей итераторов означает возможность получения значения атрибута одного объекта, в терминах другого объекта, который его представляет.

```
Interface IInputValue<T>
{
   T Value{get;}
};
interface IInputIndexer<D, T>
{
   T this[D index]{get;}
};
```

Модель итератора поддерживает записываемость [3], если на уровне моделируемой концепции определена операция допускающая изменение какого-либо атрибута представляемого объекта.

```
interface IOutputValue<T>
{
   T Value{set;}
};
public interface IOutputIndexer<D, T>
{
   T this[D index]{set;}
};
```

Каждый итератор обычно имеет ассоциированный с ним тип значения *T*, связанный с объектом-источником к которому производится обращение при выполнении операций чтения или записи.

Обобщённые концепции итераторов предоставляют возможность для естественного способа классификации, при котором алгоритмы можно разбивать по категориям в зависимости от того какие итераторы они используют. Правильный выбор концепции итератора позволяет подобрать, на этапе формирования решения вычислительной задачи, реализацию алгоритма с необходимой асимптотикой в зависимости от вида поступающих на вход данных.

6. ОБОБЩЁННЫЕ КОНЦЕПЦИИ ФУНКЦИОНАЛЬНЫХ ОБЪЕКТОВ

Большинство алгоритмов для параметризации своего поведения используют функциональные объекты, которые можно разбить на группы по типу и количеству аргументов, а также по типу возвращаемого значения. В результате, можно выделить несколько направлений, в рамках которых функциональные объекты будут задействованы как исполнители арифметических операций, операций сравнения, логических операций или операций преобразования и генерации. Есть много общего между понятием обобщённой концепции функционального объекта и понятием алгебраической операции. Если A – какое-то непустое множество, при $n \in \{0,1,\dots\}$, то тогда операцией f арности n на множестве A называется любое отображение $f: A^n \to A, (a_1, a_2, \dots, a_n) \mapsto f(a_1, a_2, \dots, a_n)$ его декартовой степени A^n в само множество A. Элементы $a_1, a_2, \dots, a_n$ называются операндами, и при $n = 0$ операция называется нульарной, при $n = 1$ – унарной, при $n = 2$ – бинарной и так далее.

Обобщённая концепция *INullaryOperation* – это базовая абстракция для операций с сигнатурой, не принимающей никаких аргументов и возвращающей вычисленный результат. Наиболее частым вариантом использования операции с нулевой арностью в обобщённых алгоритмах является генерация значений по какому-либо закону.

```
interface INullaryOperation<R>
{
   R Execute();
};
```

Обобщённая концепция *IUnaryOperation* используется как базовая абстракция для реализации в коде программ унарных операций, преобразующих единственный операнд в результат соответствующий типу возвращаемого значения.

```
interface IUnaryOperation<T, R>
{
   R Execute(T argument);
};
```

Один из возможных вариантов использования – модифицирующие алгоритмы, изменяющие элементы диапазона с помощью функционального объекта.

Обобщённая концепция *IBinaryOperation* применяется для создания операций аддитивной или мультипликативной групп, операций бинарной и битовой логики, операций над элементами множеств или более сложных, подразумевающих целый набор вложенных инструкций.

```
interface IBinaryOperation<T1, T2, R>
{
   R Execute(T1 first, T2 second);
};
```

Многие алгоритмы, для настройки своего поведения, могут использовать обобщённые концепции функциональных объектов. Основными понятиями являются: генератор (нульарная операция), отображение (унарная операция) и преобразование для множества операций с бинарной сигнатурой. Все эти направления могут быть реализованы в терминах базовых контрактов *INullaryOperation*, *IUnaryOperation*, *IBinaryOperation*, либо следовать их семантике (делегаты в C#, анонимные классы в Java). Это существенно облегчает процесс разработки и использования алгоритмов, позволяя сконцентрироваться на проектировании итерационных механизмов и основной сути того, что они должны делать.

7. РЕШЕНИЕ ЗАДАЧ МАТЕМАТИЧЕСКОГО МОДЕЛИРОВАНИЯ

Применительно к исследованию процессов и явлений в рамках какой-либо предметной области, математическое моделирование – это выявление характерных черт исследуемых объектов, определяющих их особенности, с последующей формулировкой ключевых закономерностей. Математическая модель, при этом, представляет собой систему математических соотношений, которые определяют наиболее значимые характеристики исследуемого объекта и позволяют, в рамках выбранного прикладного направления, построить реализуемое на компьютере решение. Базирующееся на современных вычислительных методах, таких как численное дифференцирование, интегрирование, и, соответственно, численное интерполирование, экстраполирование и аппроксимация, а также многих других, связанных с обработкой данных, такое решение даёт возможность проводить компьютерные эксперименты и получать на выходе результат с приемлемой точностью.

7.1. Численное интегрирование

Задача численного интегрирования возникает, когда подынтегральная функция задана не аналитическим способом или невозможно выразить аналитически её первообразную. Она состоит в замене исходной подынтегральной функции некоторым аппроксимирующим полиномом. Основой такого подхода служит вычисление конечной суммы:

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{N} C_i f(x_i), \qquad (1)$$

где C_i – числовые коэффициенты, выбор которых зависит от метода численного интегрирования, а x_i – узлы интегрирования на интервале [a, b], при $i = 1, \dots, N$.

Среди существующих способов вычисления определённых интегралов, обратим внимание на метод трапеций

$$I = \int_a^b f(x)dx \approx \frac{1}{2}(x_1 - x_0)f(a) +$$

$$+\sum_{i=1}^{N-1}\frac{1}{2}(x_{i+1} - x_{i-1})f(x_i) + \frac{1}{2}(x_N - x_{N-1})f(b) \quad (2)$$

и метод Симпсона или метод парабол

$$I = \int_a^b f(x)dx \approx \frac{1}{6}(x_1 - x_0)f(a) +$$

$$\sum_{i=1}^{N-1}\frac{1}{6}(x_{i+1} - x_{i-1})\ f(x_i) +$$

$$+\sum_{i=1}^{N-1}\frac{2}{3}(x_i - x_{i-1})f\left(\frac{1}{2}(x_i + x_{i-1})\right) +$$

$$+\frac{1}{6}(x_N - x_{N-1})f(b), \quad (3)$$

в которых, подынтегральная функция на частичном отрезке $[x_{i-1} - x_i]$ аппроксимируется интерполяционным многочленом Лагранжа первой и второй степени соответственно.

Из-за потенциальной возможности получения большого количества узлов, при попытке повышения точности расчёта, наиболее целесообразно представать их в виде последовательности индексов i с помощью итераторов, заменяя нотацию счётного диапазона ограниченным:

```
class IndexIterator :
   ForwardIterator<IndexIterator>, IInputValue<int>
{
   protected int i_;
   protected override void Advance() => ++i_;
   public IndexIterator(int i) => i_ = i;
   public virtual int Value => i_;

   /* other methods */
};
```

Основная часть программной реализации метода выносится на уровень функциональных объектов, производящих вычисления по методу трапеций

```
class Trapezoidal :
   IBinaryOperation<double, int, double>
{
   protected Func<double, double> f_;
   protected double a_, h_;
   public Trapezoidal(Func<double, double> f,
      double a, double h)
   {
      /* fields initialization */
   }
   public virtual double Execute(double t, int i)
   {
      return t +  h_ / 2 *
         (f_(a_ + i * h_) +
            f_(a_ + ( i + 1) * h_));
   }
};
```

и методу Симпсона

```
class Simpson :
   IBinaryOperation<double, int, double>
{
   protected Func<double, double> f_;
   protected double a_, h_;
   public Simpson(Func<double, double> f,
      double a, double h)
   {
      /* fields initialization */
   }
   public virtual double Execute(double s, int i)
   {
      return s + h_ / 6 * (f_(a_ + i * h_) +
         4 * f_(a_ + (i + 0.5) * h_) +
            f_(a_ + (i + 1) * h_));
   }
};
```

Все это делает возможным спецификацию поведения алгоритма без непосредственного воздействия на итерационный механизм.

В качестве основного средства обхода и объединения промежуточных результатов вычислений на отрезке $[x_{i-1} - x_i]$ и шагом $h = \frac{b-a}{N}$, используется обобщённый алгоритм Accumulate подробно рассмотренный в разделе 1.

```
var begin = new IndexIterator(0);
var end = new IndexIterator(N);
Accumulate(begin, end, 0D,
   new Trapezoidal(Math.Exp, a, b, h));
Accumulate(begin, end, 0D,
   new Simpson(Math.Exp, a, b, h));
```

Этот алгоритм является универсальным средством накопления и может выступать в роли обобщения для суммирования, произведения или каких-либо других операций, применяемых к каждому элементу диапазона с некоторым начальным значением. Стратегия организации вычислений, когда при неизменности тела алгоритма, как основного способа обработки данных, значения аргументов могут варьироваться для настройки поведения, позволяет, при необходимости, использовать другой функциональный объект, отвечающий требованиям выбранного направления.

В табл. 1 представлены результаты численного интегрирования функции e^x для различного числа узлов на интервале [0, 5] по методу трапеций и методу Симпсона.

8. ЗАКЛЮЧЕНИЕ

Для многих вычислительных задач математического моделирования от алгоритмов требуется поддержка развитой семантики при работе с данными. Степень развитости семантики для встроенного или пользовательского типа можно выразить в терминах обобщённой концепции, определяющей возможность его применения в алгоритме в ка-

Таблица 1. Результаты численного интегрирования

N	Trapezoidal [0, 5]	Simpson [0, 5]
10	150.471545997983	147.416334525476
100	147.443868897838	147.413159422459
1000	147.413466213197	147.413159102608
10000	147.413159133288	147.413159102577
100000	147.413159102879	147.413159102575
1000000	147.413159102572	147.413159102569

честве входного, выходного или внутреннего объекта. В зависимости от того на какую именно функциональность ориентирован алгоритм обобщённые концепции могут задавать: способность доступа к любой позиции обрабатываемого набора данных за константное амортизированное время; поддержку всех или одного из режимов чтения или записи значений элементов; возможность обхода с постоянным шагом в прямом и обратном направлениях; сохранение текущего состояния с помощью создания глубокой (deep) или поверхностной (shallow) копии; применимость в выражениях требующих поддержки унарных или бинарных операций арифметики, логики, различных вариантов сравнения.

В качестве слоя абстракций для представления и обработки данных в статье рассмотрены обобщённые концепции итераторов и функциональных объектов. Их совместное применение создаёт методологически целостную основу, обеспечивающую надлежащий уровень гибкости при разработке и использовании обобщённых алгоритмов. Расширяя границы и возможности научного программирования, предлагаемый подход позволяет создавать решения для широкого класса вычислительных задач на объектно-ориентированном языке.

СПИСОК ЛИТЕРАТУРЫ

1. *Ортега, Дж.* Введение в численные методы и решения дифференциальных уравнений / Дж. Ортега; пер. с англ. – М.: Наука, 1986. – 288 с.
2. *Fowler, M.* Domain-Specific Languages / M. Fowler. – Boston: Addison–Wesley, 2010. – 640 p.
3. *Booch, G.* Object-oriented analysis and design / G. Booch. – California: Addison–Wesley, 1998. – 534 p.
4. *Cardelli, L.* A Semantics of Multiple Inheritance / L. Cardelli // Information and Computation. –1988. – Vol. 76. – P. 138-164.
5. *Bloom, B.S.* Taxonomy of Educational Objectives: The Classification of Educational Goals / B.S. Bloom. – Chicago: Longmans, 1956. – 207 p.
6. *Yablokova, L.V.* The concept of «range» used in experimental calculations / L.V. Yablokova // CEUR Workshop Proceedings. – 2015. – Vol. 1490. – P. 402-405.
7. *Вандевурд, Д.* Шаблоны C++: справочник разработчика / Д. Вандевурд, Н. Джосаттис. – М.: Вильямс, 2003. – 538 с.
8. *Siek, J.* Concept checking: Binding parametric polymorphism in C++. First Workshop on C++ Template Programming / J. Siek, A.Lumsdaine // ACM Java Grande–ISCOPE. – 2005. – Vol.17, Issue7-8. – P. 941– 965.
9. *Austern, M.H.* Generic Programming and the STL: Using and Extending the C++ Standard Template Library / M.H. Austern – Boston: Addison–Wesley, 1999. – 304 p.
10. *Kuhn, T. S.* The structure of scientific revolutions / T.S. Kuhn. – Chicago: University Press, 1962. – 264 p.
11. *Floyd, R. W.* The paradigms of programming / R. W. Floyd // Communications of the ACM. – 1979. – Vol. 22, Issue 8. – P. 455-460.
12. *Jarvi, J.* Associated types and constraint propagation for mainstream object-oriented generics / J. Jarvi, J. Willcock, A. Lumsdaine // OOPSLA, October. – 2005.
13. *Stepanov, A.* The Standard Template Library / A. Stepanov, M. Lee. – Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, 1994 http://www.hpl.hp.com/ techreports.
14. *Stepanov, A.* Elements of Programming / A. Stepanov, P. McJones. – Boston: Addison–Wesley, 2011. – 272 p.
15. *Breymann, U.* Designing Components with the C++STL / U. Breymann. – Bonn: Addison–Wesley, 2000. – 301 p.
16. *Stroustrup, B.* Concept Design for the STL / B. Stroustrup, A. Sutton – Luxembourg: Springer Link, 2012. – 133 p.

USING OF GENERIC CONCEPTS OF ITERATORS AND FUCTIONAL OBJECTS DURING THE SOLVING OF MATHEMATICAL MODELING PROBLEMS IN OBJECT-ORIENTED LANGUAGES WITH SUPPORT OF F-BOUNDED POLIMORPHISM

© 2018 D.E. Yablokov

Samara National Research University named after Academician S.P. Korolyov

The article deals with an alternatively approach to scientific programming, based on the conceptual-holistic methodology that combines the features of several programming paradigms. When solving mathematical modeling problems related to physical research in optics, the use of pure object-oriented languages, such as C# and Java is quite rare. Excepting performance questions, it is connected to the underdeveloped means of iteration procedure and access to elements of collections. Also it is connected to the lack of complete information about the concrete type in the universal declarations of methods and abstract data types in compile time. For elimination of these shortcomings in the article use of the generic concepts allowing specifying requirements to arguments of algorithms is offered. The ability to control execution preconditions based on syntax and semantic restrictions sets the required strategy of data processing. The concepts is widely known in the C ++ community and means the requirement specification to parameters of types in the predicative form. These requirements set rules of application of syntax constructions and give understanding of what types, literals, valid operations, and expressions must be associated with them. Article is intended for permanently the practicing experts using C ++, C# or Java in scientific work and will be interesting from the point to combining of many, already existing, concepts, methods and technologies of data processing.
Keywords: Generic algorithm, generic concept, range notation, traversal of range, iterator concept, operation concept, object-oriented language, F-bounded polymorphism.

Denis Yablokov, Senior Developer of Samara Center for Theoretical Materials Science. E-mail: dyablokov@gmail.com