Концепция полиморфизма и ее реализация в языке C#

Комментарий к слайду

В данной лекции будут рассмотрены вопросы, относящиеся к истории развития, идеологии, математическому основанию и обзору возможностей полиморфизма – одной из фундаментальных концепций, на которых основано объектно-ориентированное программирование.

Содержание лекции

Комментарий к слайду

Коротко о содержании лекции.

В ходе лекции будут рассмотрены важнейшие научные исследования, относящиеся к эволюции подходов к математическому моделированию одной из важнейших для объектно-ориентированного подхода к программированию концепций аспекта, а именно, полиморфизма.

Прежде всего, будет сформулировано определение понятия полиморфизма.

Затем будет представлен сравнительный анализ путей реализации концепции полиморфизма в языках функционального и объектно-ориентированного программирования и в computer science.

При этом будут подробно исследованы особенности различных видов полиморфизма.

Особое внимание будет уделено реализации механизмов реализации абстрактных типов данных и вызова процедур в языке программирования C#, включая вызов по имени, по ссылке, а также «ленивое» означивание при вызове по необходимости.

Фрагменты программ на языках SML и C# проиллюстрируют практику применения концепции полиморфизма и дадут возможность сопоставления особенностей ее реализации в зависимости от подхода.

Лекция завершится обзором литературы для более глубокого исследования материала.

Основные результаты исследований полиморфизма

1934 – А. Черч (Alonso Church) изобрел ламбда-исчисление и исследовал порядок вычислений в ламбда-термах

1936 – Г. Плоткин (G.D. Plotkin) исследовал стратегии вызова по имени и по значению на основе ламбда-исчисления

1960's – П.Лендин (Peter J. Landin) создал SECD-машину, математический формализм, моделирующий вызов по имени

1969 – Р.Хиндли (Roger Hindley) исследовал полиморфные системы с типами

1978 –Р.Милнер (Robin Milner) предложил расширенную систему полиморфной типизации для языка программирования ML

1989-90 – У.Кук, П.Кэннинг, У.Хилл и др. (William R. Cook, Peter S. Canning, Walter L. Hill et al.) исследовали полиморфизм в ООП и его связь с ламбда-исчислением

Комментарий к слайду

Кратко остановимся на наиболее значительных (с точки зрения данного курса) этапах эволюции теории и практики реализации полиморфизма в формальных теориях и языках программирования.

В 30-х г.г. А. Черч (Alonso Church) предложил исчисление ламбда-конверсий или ламбда-исчисление и применил его для исследования теории множеств. Вклад ученого был настолько фундаментальным, что теория (до сих пор называемая ламбда-исчислением и часто именуемая в литературе ламбда-исчислением Черча) является основополагающей и для рассматриваемых нами вопросов.

Исследования различных стратегий передачи параметров при обращении к (полиморфным) функциям языков программирования (в частности, вызова функций по имени и по значению) на основе ламбда-исчисления были проведены Г. Плоткиным (G.D. Plotkin). Заметим, что полученные результаты значительно позднее (уже в 70-х г.г.) были использованы для моделирования вычислений в ранних версиях языка функционального программирования ML.

В 60-х г.г., уже в эпоху высокоуровневых языков программирования, П. Лендином (Peter J. Landin) была разработана так называемая SECD-машина, а именно, математическая формализация для реализации вычислений в терминах ламбда-выражений. Кроме того, тем же автором был создан формальный язык ISWIM (If you See What I Mean), представляющий собой вариант расширенного ламбда-исчисления и ставший впоследствии прообразом языка программирования ML.

В конце 60-х г.г. Р. Хиндли (Roger Hindley) исследовал полиморфные системы типов, т.е. такие системы типов, в которых возможны параметризованные функции или функции, имеющие переменный тип. При этом основной проблемой, стоящей перед исследователем, было моделирование языков программирования со строгой типизацией.

Затем, в 70-х г.г. Р. Милнер (Robin Milner) предложил практическую реализацию расширенной системы полиморфной типизации для языков функционального программирования.

Наконец, на рубеже 80-х и 90-х г.г., рядом исследователей – У.Куком (William R. Cook), П.Кэннингом (Peter S. Canning), У.Хиллом (Walter L. Hill) и другими – была изучена концепция полиморфизма в приложении к объектно-ориентированному программированию (в частности, к языку C++) и выявлена возможность моделирования полиморфизма в ООП на основе ламбда-исчисления.

Понятие полиморфизма в программировании

Вообще говоря, под ***полиморфизмом*** понимается возможность оперировать объектами, не обладая точным знанием их типов.

В функциональном программировании и ООП понятие полиморфизма связано с:

- наследованием;
- интерфейсами;
- отложенным связыванием (“ленивыми” или “замороженными” вычислениями)

Комментарий к слайду

По рассмотрении теоретических оснований такой важнейшей концепции объектно-ориентированного программирования как полиморфизм, предпримем попытку формализовать это понятие.

Под полиморфизмом будем иметь в виду возможность оперирования объектами без однозначной идентификации их типов.

Напомним, что понятие полиморфизма уже исследовалось в части курса, посвященной функциональному подходу к программированию.

Наметим концепции, объединяющие функциональный и объектно-ориентированный подходы к программированию с точки зрения полиморфизма.

Как было отмечено в ходе исследования функционального подхода к программированию, концепция полиморфизма предполагает в части реализации отложенное связывание переменных со значениями. При этом во время выполнения программы происходят так называемые “ленивые” или, иначе, “замороженные” вычисления. Таким образом, означивание языковых идентификаторов выполняется по мере необходимости.

В случае объектно-ориентированного подхода к программированию теоретический и практический интерес при исследовании концепции полиморфизма представляет отношение наследования, прежде всего, в том смысле, что это отношение порождает семейства полиморфных языковых объектов.

С точки зрения практической реализации концепции полиморфизма в языке программирования C# в форме полиморфных функций особую значимость для исследования представляет механизм интерфейсов.

Полиморфизм типов в языке SML

Встроенная функция `hd` для списка произвольного типа:

```
hd [1, 2, 3];
val it = 1: int
```
(тип функции: (int list) → int)

```
hd [true, false, true, false];
val it = true: bool
```
(тип: (bool list) → bool)

```
hd [(1,2)(3,4),(5,6)];
val it = (1,2) : int*int
```
((int*int)list→(int*int))

Функция `hd` имеет тип (*type* `list`) →*type*, где *type* – произвольный тип

Комментарий к слайду

Напомним, что реализация полиморфизма при функциональном подходе к программированию основана на оперировании функциями переменного типа.

Для иллюстрации исследуем поведение встроенной функции `hd`, (от слова «head» – голова), которая выделяет «голову» (первый элемент) списка, вне зависимости от типа его элементов. Применим функцию к списку из целочисленных элементов:

```
hd [1, 2, 3];
val it = 1: int
```

Получим, что функция имеет тип функции из списка целочисленных величин в целое число (`int list → int`). В случае списка из значений истинности та же самая функция

```
hd [true, false, true, false];
val it = true: bool
```

возвращает значение истинности, т.е. имеет следующий тип: `bool list → bool`. Наконец, для случая списка кортежей из пар целых чисел

```
hd [(1,2)(3,4),(5,6)];
val it = (1,2) : int*int
```

получим тип `((int*int)list→ (int*int))`. В итоге можно сделать вывод о том, что функция `hd` имеет тип (*type* `list`) → *type*, где *type* – произвольный тип, т.е. полиморфна.

Полиморфизм типов в языке C#

Рассмотрим пример полиморфной функции:

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

а также примеры ее применения:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```


Комментарий к слайду

Проиллюстрируем сходные черты и особенности реализации концепции полиморфизма при функциональном и объектно-ориентированном подходе к программированию следующим примером фрагмента программы на языке C#:

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

Как видно, приведенный пример представляет собой описание полиморфной функции `Poly`, которая выводит на устройство вывода (например, на экран) произвольный объект `o`, преобразованный к строковому формату (`o.ToString()`).

Рассмотрим ряд примеров применения функции `Poly`:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```

Заметим, что независимо от типа аргумента (в первом случае это целое число `25`, во втором – символьная строка `"John Smith"`, в третьем – вещественное число π=`3.141592536`, в четвертом – объект типа `Point`, т.е. точка на плоскости с

координатами `(12,45)`), обработка происходит единообразно и, как и в случае с языком функционального программирования SML, функция генерирует корректный результат.

Понятия, связанные с полиморфизмом в C#

1. Интерфейсы (функции с множеством элементов; четко определенные соглашения)
2. Типы (определяют на интерфейсы объектов и их реализацию; переменные также могут быть типизированными)
3. Наследование (классы и типы объединяются в иерархии базовых (или надклассов) и производных (или подклассов). Существуют определенные различия между наследованием интерфейсов и реализаций
4. Отложенное связывание или ленивые вычисления (значения присваиваются (связываются) объектам по мере того, как они требуются во время выполнения программы)

Комментарий к слайду

Как видно из приведенных выше примеров, концепция полиморфизма одинаково применима как к функциональному, так и к объектно-ориентированному подходу к программированию. При этом целью полиморфизма является унификация обработки разнородных языковых объектов, которые в случае функционального подхода являются функциями, а в случае объектно-ориентированного – объектами переменного типа. Отметим, что для реализации полиморфизма в языке объектно-ориентированного программирования C# требуется четкое представление о ряде понятий и механизмов.

Естественно, что говорить о полиморфизме можно только с учетом понятия типа. Типы определяют интерфейсы объектов и их реализацию. Переменные, функции и объекты в рассматриваемых в данном курсе языках программирования также рассматриваются как типизированные элементы. Необходимо также напомнить, что важное практическое значение при реализации полиморфизма в языке C# имеет механизм интерфейсов (под которыми понимаются чисто абстрактные классы с поддержкой полиморфизма, содержащие только описания без реализации). Для реализации концепции множественного наследования необходимо принять ряд дополнительных соглашений об интерфейсах.

Сложно говорить о полиморфизме и в отрыве от концепции наследования, при принятии которой классы и типы объединяются в иерархические отношения частичного порядка из базовых классов (или, иначе, надклассов) и производных классов (или, иначе, подклассов). При этом существуют определенные различия между наследованием интерфейсов как частей, отвечающих за описания классов и частей, описывающих правила реализации.

Еще одним значимым механизмом, сопряженным с полиморфизмом, является так называемое отложенное связывание (или, иначе, ленивые вычисления), в ходе которых

значения присваиваются объектам (т.е. связываются с ними) по мере того как, эти значения требуются во время выполнения программы.

Методы вызова процедур

1. Вызов по значению – call-by-value (в числе первых моделей computer science – абстрактная SECD-машина П.Лендина (Peter J.Lendin))

2. Вызов по имени – call-by-name (или вызов по ссылке – call-by-reference)

3. Вызов по необходимости – call-by-need («ленивые» – “lazy”, «замороженные» –“frozen” или отложенные вычисления)

Комментарий к слайду

Напомним классификацию стратегий вычислений, построенную в части курса, посвященной исследованию функционального подхода к программированию.

При вычислении с вызовом по значению (call-by-value) все выражения должны быть означены до вычисления операции аппликации. Заметим, что формализация стратегии вычислений с вызовом по значению возникла в числе первых моделей computer science в виде абстрактной SECD-машины П.Лендина (Peter J.Lendin).

При вычислении с вызовом по имени (call-by-name) до вычисления операции аппликации необходима подстановка термов вместо всех вхождений формальных параметров до означивания. Стратегию вычислений с вызовом по значению иначе принято называть вызовом по ссылке (call-by-reference).

Наконец, при вычислении с вызовом по необходимости (call-by-need) ранее вычисленные значения аргументов сохраняются в памяти компьютера только в том случае, если необходимо их повторное использование. Именно эта стратегия лежит в основе «ленивых» (lazy), «отложенных» (delayed) или «замороженных» (frozen) вычислений, которые принципиально необходимы для обработки потенциально бесконечных структур данных.

Вызов по значению

В случае ***вызова по значению*** (call-by-value, CBV):

1) формальный параметр является копией фактического параметра;
2) фактический параметр является выражением

Пример использования (ввод значений)

```
void Inc(int x) {x = x + 1;}
void f() {
  int val = 3;
  Inc(val); // val == 3
}
```


Комментарий к слайду

Рассмотрим более подробно особенности стратегии вычислений с вызовом по значению (call-by-value, CBV).

Прежде всего, в случае вызова по значению формальный параметр является копией фактического параметра и занимает выделенную область в памяти компьютера.

Кроме того, фактический параметр в случае вызова по значению является выражением.

Проиллюстрируем особенности использования стратегии вызова по значению следующим фрагментом программы на языке C#:

```
void Inc(int x) {x = x + 1;}
void f() {
     int val = 3;
     Inc(val); // val == 3
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует ввод значений `val` посредством функции `f`. Функция `Inc` является функцией следования.

Отметим, что несмотря на применение функции `Inc` к аргументу `val`, значение переменной `val`, как явствует из комментария `// val == 3`, остается неизменным. Это обусловлено тем обстоятельством, что при реализации стратегии вызова по значению формальный параметр является копией фактического.

Вызов по имени (ссылке)

В случае ***вызова по имени*** (иначе по ссылке, или call-by-reference, CBR):

1) формальный параметр является подстановкой (alias) фактического параметра (передается адрес фактического параметра);

2) фактический параметр должен быть переменной, формальный параметр является копией фактического параметра

Пример (преобразование значений):

```
void Inc(ref int x) { x = x + 1; }
void f() {
  int val = 3;
  Inc(ref val); // val == 4
}
```


Комментарий к слайду

Рассмотрим более подробно особенности стратегии вычислений с вызовом по имени, или, иначе, по ссылке (call-by-reference, CBR).

Прежде всего, в случае вызова по имени формальный параметр является подстановкой (alias) фактического параметра и не занимает отдельной области в памяти компьютера. При реализации данной стратегии вычислений вызывающей функции передается адрес фактического параметра.

Кроме того, фактический параметр в случае вызова по имени должен быть не выражением, а переменной. При этом формальный параметр является копией фактического параметра.

Проиллюстрируем особенности использования стратегии вызова по имени следующим фрагментом программы на языке C#:

```
void Inc(ref int x) { x = x + 1; }
void f() {
     int val = 3;
     Inc(ref val); // val == 4
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует преобразование значений `val` посредством функции `f`. Функция `Inc` является функцией следования.

Отметим, что вследствие применения функции `Inc` к аргументу `val`, значение переменной `val`, как явствует из комментария `// val == 4`, изменяется. Это обусловлено тем обстоятельством, что при реализации стратегии вызова по имени формальный параметр является подстановкой фактического.

Вызов по необходимости

В случае ***вызова по необходимости*** (call-by-need, CBN):

1) ситуация с параметром аналогична CBR, однако значение не передается вызывающей функции;
2) не следует использовать в методах до того, как значение будет получено

Пример (ввод значений):

```
void Read (out int first, out int next) {
  first = Console.Read(); next = Console.Read();
}
void f() {
  int first, next;
  Read(out first, out next);
}
```


Комментарий к слайду

Рассмотрим более подробно особенности стратегии вычислений с вызовом по необходимости (call-by-need, CBN).

В целом, данная стратегия вычислений сходна вызовом по имени (или ссылке, call-by-reference, CBR), однако имеет ее реализация имеет две характерные особенности.

Во-первых, в случае вызова по необходимости значение фактического параметра не передается вызывающей функции, т.е. не происходит связывания переменной со значением.

Во-вторых, данная стратегия вычислений неприменима до того, как означивание может быть произведено, т.е. значение фактического параметра может быть получено.

Проиллюстрируем особенности использования стратегии вызова по необходимости следующим фрагментом программы на языке C#:

```
void Read (out int first, out int next) {
     first = Console.Read(); next = Console.Read();
}
void f() {
     int first, next;
     Read(out first, out next);
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует ввод значений `first` и `next` посредством функции `Read` со стандартного устройства ввода. Функция `f` может быть означена по необходимости, по мере поступления аргументов.

Абстрактные классы и методы в языке C#

1.Средство реализации концепции полиморфизма.

2.Абстрактные методы не имеют части реализации (implementation).

3. Абстрактные методы неявно являются виртуальными (virtual).

4. В случае, если класс имеет абстрактные методы, его необходимо описывать как абстрактный.

5. Запрещено создавать объекты абстрактных классов.

Комментарий к слайду

Исследовав особенности реализации различных стратегий вычислений в языке программирования C#, рассмотрим концепцию полиморфизма в соотнесении с механизмом так называемых абстрактных классов.

Абстрактные классы при объектно-ориентированном подходе (в частности, в языке программирования C#) являются аналогами полиморфных функций в языках функционального программирования (в частности, в языке SML) и используются для реализации концепции полиморфизма. Методы, которые реализуют абстрактные классы, также называются абстрактными и являются полиморфными.

Перечислим основные особенности, которыми обладают абстрактные классы и методы в рамках объектно-ориентированного подхода к программированию.

Прежде всего, отметим то обстоятельство, что абстрактные методы не имеют части реализации (implementation).

Кроме того, абстрактные методы неявно являются виртуальными (т.е. как бы оснащенными описателем `virtual`).

В том случае, если внутри класса имеются определения абстрактных методов, данный класс необходимо описывать как абстрактный. Ограничений на количество методов внутри абстрактного класса в языке программирования C# не существует.

Наконец, в языке программирования C# запрещено создание объектов абстрактных классов (как конкретизаций или экземпляров).

Абстрактные классы в языке C#: пример применения

```
abstract class Stream {
  public abstract void Write(char ch);
  public void WriteString(string s) {
        foreach (char ch in s) Write(s);
 }
}

class File : Stream {
  public override void Write(char ch) {
        ... write ch to disk ...
  }
}
```


Комментарий к слайду

Проиллюстрируем особенности использования абстрактных классов следующим фрагментом программы на языке C#:

```
abstract class Stream {
          public abstract void Write(char ch);
          public void WriteString(string s) {
               foreach (char ch in s) Write(s);
      }
     }
     class File : Stream {
          public override void Write(char ch) {
               ... write ch to disk ...
          }
     }
```

Как видно из приведенного примера, фрагмент программы на языке C# представляет собой описание абстрактных классов `Stream` и `File`, реализующих потоковую запись (метод `Write`) данных в форме символьных строк `ch`.

Заметим, что описание абстрактного класса `Stream` реализовано явно посредством зарезервированного слова `abstract`. Оператор `foreach … in` реализует последовательную обработку элементов и будет рассмотрен более подробно в продолжение курса.

Необходимо обратить внимание на то обстоятельство, что поскольку в производных классах необходимо замещение методов, метод `Write` класса `File` оснащен описателем `override`.

Преимущества концепции полиморфизма

1. Унификация обработки объектов различной природы
2. Снижение стоимости программного обеспечения
3. Повторное использование кода
4. Интуитивная прозрачность исходного текста
4. Строгое математическое основание (ламбда-исчисление)
5. Концепция является универсальной и в равной степени применима в функциональном и объектно-ориентированном программировании

Комментарий к слайду

Подводя итоги рассмотрения основных аспектов концепции полиморфизма в объектно-ориентированном подходе к программированию и особенностей реализации этой концепции применительно к языку программирования C#, кратко отметим достоинства полиморфизма.

Прежде всего, к преимуществам концепции полиморфизма следует отнести унификацию обработки объектов различной природы. В самом деле, абстрактные классы и методы позволяют единообразно оперировать гетерогенными данными, причем для адаптации к новым классам и типам данных не требуется реализации дополнительного программного кода.

Кроме того, важным практическим следствием реализации концепции полиморфизма для экономики программирования является снижение стоимости проектирования и реализации программного обеспечения.

Еще одним существенным достоинством полиморфизма является возможность усовершенствования стратегии повторного использования кода. Код с более высоким уровнем абстракции не требует существенной модификации при адаптации к изменившимся условиям задачи или новым типам данных.

Важно также отметить, что идеология полиморфизма основана на строгом математическом фундаменте (в частности, в виде формальной системы ламбда-исчисления), что обеспечивает интуитивную прозрачность исходного текста для математически мыслящего программиста, а также верифицируемость программного кода.

Наконец, концепция полиморфизма является достаточно универсальной и в равной степени применима для различных подходов к программированию, включая функциональный и объектно-ориентированный.

Библиография (1)

1. Pratt T.W., Zelkovitz M.V. Programming languages, design and implementation (4th ed.).- Prentice Hall, 2000
2. Appleby D., VandeKopple J.J. Programming languages, paradigm and practice (2nd ed.).- McGraw-Hill, 1997
3. Milner R. A theory of type polymorphism in programming languages. Journal of Computer and System Science, 17(3):348-375, 1978
4. Canning P.S., Cook W.R., Hill W.L., Olthoff W., Mitchell J.C. F-bounded polymorphism for object-oriented programming.Conference on Functional Programming and Computer Architecture, 1989, p.p. 273-280

Комментарий к слайду

К сожалению, в рамках времени, отведенных на одну лекцию, можно лишь в общих чертах охарактеризовать специфику такой фундаментальной концепции объектно-ориентированных языков, ООП, а также для целого ряда других подходов к программированию как полиморфизм. Рассмотрение более глубоких аспектов полиморфизма планируется в продолжение курса. Для более детального ознакомления с особенностями, достижениями и проблемами в теории моделирования этой концепции и практики реализации связанных с ней механизмов рекомендуется следующий список литературы:

1. Pratt T.W., Zelkovitz M.V. Programming languages, design and implementation (4th ed.).- Prentice Hall, 2000

2. Appleby D., VandeKopple J.J. Programming languages, paradigm and practice (2nd ed.).- McGraw-Hill, 1997

3. Milner R. A theory of type polymorphism in programming languages. Journal of Computer and System Science, 17(3):348-375, 1978

4. Canning P.S., Cook W.R., Hill W.L., Olthoff W., Mitchell J.C. F-bounded polymorphism for object-oriented programming. Conference on Functional Programming and Computer Architecture, 1989, p.p. 273-280

Кратко остановимся на источниках. В работе [1] приведен наиболее полный анализ истории развития и особенностей языков программирования с классификацией по областям применения. В работе [2] рассмотрены вопросы проектирования и реализации современных языков программирования. Работа [3] представляет собой исследование концепции полиморфизма в условиях функционального подхода к программированию.

Работа [4] посвящена исследованию применимости полиморфизма в функциональном программировании к ООП.

Библиография (2)

5. Thorup L., Tofte M. Object-oriented programming and Standard ML. Proc. ACM SIGPLAN 1994 Workshop on ML and its applications, Orlando, FL, June 1994, Tech. Report 2265 INRIA, p.p. 41-49.
6. Troelsen A. C# and the .NET platform (2nd ed.).- APress, 2003, 1200 p.p.
7. Liberty J. Programming C# (2nd ed.).- O'Reilly, 2002, 656 p.p.
8. Plotkin G.D. Call-by-name, call-by-value and the λ-calculus. Theoretical computer science, 1, pp. 125-159, 1936
9. Turner D.A. A new implementation technique for applicative languages. Software – Practice and Experience, 9:21-49, 1979

Комментарий к слайду

Продолжим обсуждение работ, посвященных исследованию теоретических основ и практической реализации концепции полиморфизма.

5. Thorup L., Tofte M. Object-oriented programming and Standard ML. Proc. ACM SIGPLAN 1994 Workshop on ML and its applications, Orlando, FL, June 1994, Tech. Report 2265 INRIA, p.p. 41-49

6. Troelsen A. C# and the .NET platform (2nd ed.).- APress, 2003, 1200 p.p.

7. Liberty J. Programming C# (2nd ed.).- O'Reilly, 2002, 656 p.p.

8. Plotkin G.D. Call-by-name, call-by-value and the λ-calculus. Theoretical computer science, 1, pp. 125-159, 1936

9. Turner D.A. A new implementation technique for applicative languages. Software – Practice and Experience, 9:21-49, 1979

Продолжим обсуждение библиографии. Работа [5] анализирует взаимосвязи полиморфизма в рамках объектно-ориентированного и функционального подхода к программированию и иллюстрирует их примерами программ на языках SML и C++. В работе [6] рассмотрены теоретические проблемы и практические аспекты реализации инновационных конструкций в языках программирования, прежде всего, в языке C#.NET, изучение которого составляет основу курса. В работе [7] рассматриваются вопросы, связанные с разработкой программ на языке C#. Работа [8] представляет собой сравнительное исследование стратегий вычислений в современном программировании. Работа [9] посвящена формализации полиморфизма средствами теоретического computer science.