МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И.И. Воровича

Волошин Богдан Иванович

РЕАЛИЗАЦИЯ КЛАССОВ ТИПОВ ДЛЯ PASCALABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению 02.04.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доцент, к.ф.-м.н. Михалкович Станислав Станиславович

Рецензент –
доцент, к.т.н. Демьяненко Яна Михайловна

Ростов-на-Дону – 2018

СОДЕРЖАНИЕ

Введение

Вычислительные машины получили широкое распространение в профессиональной и личной жизни каждого индивидуума в цивилизованном обществе. Сфера информационных технологий уже давно являет нечто большее, чем узкий круг профессиональных интересов отдельных специалистов. Автоматизация процессов производства, как естественный процесс в постиндустриальном обществе базируется на достижениях науки в данной области. Развитие информационных технологий явилось основополагающей причиной процесса глобализации, особенно явно это видно на примере культурной унификации за счёт глобальной сети Интернет. С развитием технологий накапливаются массивы данных, а значит и растёт сложность задач, это в свою очередь порождает новые подходы и новые задачи, что в итоге приводит к расширению границ решаемых задач. Всё чаще информационные технологии находят применение в областях изначально не пригодных для информационных технологий, например, в социальной сфере отсутствие возможности постановки строго формализованных задач породило развитие эвристических методик. Отражение этого факта можно видеть в огромном количестве разнообразного программного обеспечения, которое решает крупные, многосоставные задачи.

В свою очередь специалист в области информационных технологий является не только потребителем программного обеспечения, но и производителем оного. Поэтому важно понимать, что программное обеспечение является интеллектуальной собственностью, и важным механизмом регулировки отношений между пользователями и потребителями программного обеспечения является лицензия. Существует два основных направления лицензирования: несвободное и свободное. В первом случае потребитель не является владельцем копии программного обеспечения. Во втором случае наоборот. В данной работе используется продукт компании Microsoft – Visual Studio, которая используется в соответствии с

предоставленной Институтом Механики Математики и Компьютерных наук им. Воровича подпиской Microsoft Imagine (ранее – MSDN).

Данная магистерская работа посвящена реализации классов типов для PascalABC.NET. PascalABC.NET – современный язык программирования, основанный на языке Object Pascal [1]. Проект реализуется на объектно-ориентированном языке C# [16]. Язык PascalABC.NET предоставляет множество современных языковых средств [2]. Одним из таких средств являются обобщённые классы и подпрограммы [5]. Обобщённое программирование — это важный и часто используемый подход для разработки программного обеспечения, который позволяет избегать повторения участков кода. Примером использования этого подхода может служить полиморфный контейнер. Большинство современных языков программирования предоставляют разные языковые средства для использования обобщённого программирования.

В данный момент обобщённое программирование в языке PascalABC.NET реализуется с помощью обобщённых типов. Аналогичный подход используется в языке программирования C#. В данном подходе обобщённые алгоритмы или классы могут накладывать ограничения на параметры обобщения, при этом ограничения могут быть только ограничением наследования класса или интерфейса.

Важно отметить, что помимо упомянутого выше языкового средства существуют другие способы реализации обобщённого программирования. В функциональном языке программирования Haskell используются классы типов. Классы типов не являются типами, они представляют собой категории типов. В отличие от классов в объектно-ориентированных языках программирования, классы типов представляют независимую таблицу виртуальных методов. Это позволяет отделить реализацию типа от указания каким ограничениям удовлетворяет этот тип.

У обобщённых типов есть ряд недостатков. Например, у интерфейсов нет реализации по умолчанию, в отличие от классов типов, которые

предоставляют такую возможность. Кроме того, обобщённые типы многословны. Например, если в интерфейсе есть типовый параметр, на который накладывается ограничение, то во всех обобщённых алгоритмах, необходимо указывать это ограничение явно, что приводит к разрастанию кода.

Проведённое исследование относится к кругу задач связанных с компиляторостроением. В процессе проведения исследований были учтены выпускные магистерские работы по схожим направлениям. В частности работа Алгасова А.С. по реализации паттерн матчинга. С автором данной работы было проведено обсуждение, нацеленное на унификацию работ в рамках проекта PascalABC.NET. Для организации совместной работы была использована система контроля версий github [19].

Постановка задачи

1. Реализовать классы типов для языка PascalABC.NET с поддержкой следующих возможностей:
   - Множественное наследование классов типов
   - Реализация по умолчанию для подпрограмм определённых в классе типов
2. Обеспечить поддержку перегрузки операторов:
   - Переопределение операторов внутри класса типа для элементов произвольного типа
   - Поиск операторов в классе типов, если таковых нет во внешнем контексте
3. Обеспечить поиск подпрограммы в классе типов, если явно не указано, что подпрограмма принадлежит классу типу, и при этом нет подходящей подпрограммы во внешнем контексте
4. Модифицировать алгоритм вывода шаблонных параметров для поиска необходимого инстанса, при инстанцировании обобщённых подпрограмм на которых наложено ограничение классами типов.

Глава1. Обобщённое программирование в современных языках программирования

1.1 Определение классов типов

Классы типов – языковое средство, предоставляющее возможности для применения подхода обобщённого программирования, основанное на ad-hoc полиморфизме.

Обобщённое программирование — это подход к написанию программного обеспечения, который заключается в том, что типы данных и алгоритмы параметризированы типами и могут быть использованы с различными типами данных. Все современные языки программирования предоставляют средства обобщённого программирования. Этот подход основан на абстрактных описаниях требований. То есть описывается не конкретный тип, а семейство типов. При этом каждый тип из этого семейства имеет схожее поведение.

В свою очередь ad-hoc полиморфизм – это разновидность полиморфизма, при котором обобщённые функции могут быть применены к аргументам разных типов, за счёт перегрузки функций и операций.

В случае классов типов, определяются требования к типу в виде функций. А потом для каждого типа, удовлетворяющего этим требованиям, перегружаются соответствующие функции (см. Пример 1).

```
type
  Sometypeclass[TemplateType] = typeclass
    function EverySometypeclassShouldHaveThisFunction(
                  value: TemplateType): ReturnType;
    function AndThisFunctionIsImportantToo(
                  someNumber: integer,
                  restrictedValue: TemplateType): ReturnType;
    …
  end;

  Sometypeclass[InstanceType] = instance
    function EverySometypeclassShouldHaveThisFunction(
                  value: TemplateType): ReturnType;
    begin
      Result := ImplementationForInsanceType(value);
    end;
```

```
    function AndThisFunctionIsImportantToo(
                  someNumber: integer,
                  restrictedValue: TemplateType): ReturnType;
    begin
      Result := AnotherImplementationForInstanceType(
                  someNumber, restrictedValue);
    end;
    …
  end;
```

Пример 1

При реализации алгоритма можно потребовать, чтобы тип принадлежал к нужной категории, а значит реализовывал необходимые функции. В примере 2 показано, как, используя код из примера 1, реализовать обобщённый алгоритм с требованием к параметру обобщения. Из примера видно, что к переменной, имеющей тип шаблонного параметра, применяются функции, описанные в классе типа. При этом при вызове этого алгоритма, если типовый параметр будет удовлетворять требованиям класса типа, то будут вызываться функции, описанные в соответствующей перегрузке.

```
function GenericFunction<FunctionTemplateType>(
           value: FunctionTemplateType): ReturnType;
         where Sometypeclass[FunctionTemplateType];
begin
  var resultOfFunction1 :=
        EverySometypeclassShouldHaveThisFunction(value);
  var resultOfFunction2 :=
        AndThisFunctionIsImportantToo(777, value);
  …
end;

begin
  var value: InstanceType;
  …
  GenericFunction(value);
end.
```

Пример 2

Более подробное описание возможностей классов типов, а также в частности реализованные возможности в языке PascalABC.NET, включая синтаксис конструкций приведено в последующих разделах.

1.2 Основные понятия, связанные с классами типов

В этом разделе рассматриваются понятия, возникающие при использовании классов типов, а также разобраны основные языковые конструкции, необходимые для классов типов.

1.2.1 Класс типа, как часть языкового средства

Рассматриваемый подход принято называть классами типов, но класс типов является только одной из языковых конструкций, необходимых для реализации обобщённого программирования. Сам класс типов можно рассматривать, как набор ограничений, которые определяют не только множество типов, но и то, как эти типы можно однообразно использовать.

В качестве ограничений зачастую выступают функции над определяемой категорией типов. И каждый тип, принадлежащий этому классу типов, должен реализовывать эту функцию.

Определение из примера 3 читается, как «некий тип является группой, если над этим типом определена операция умножения, при чём результат умножения принадлежит тому же типу, есть единичный элемент и для каждого элемента этого типа можно получить обратный ему элемент».

```
type
  Group[SomeType] = typeclass
    function operator*(a, b: SomeType): SomeType;
    function IdentityElement(): SomeType;
    function GetInverseElement(a: SomeType): SomeType;
  end;
```

Пример 3

1.2.2 Наследование классов типов

При определении класса типов зачастую возникает необходимость использовать определённые ранее ограничения. Чтобы избежать дублирования кода, вводится понятие наследования классов типов. Стоит отметить, что, в отличие от классов в объектно-ориентированном программировании, классы типов имеют множественное наследование. Интерфейсы – ближайшая аналогия из объектно-ориентированного программирования для классов типов.

В данной работе рассматривается не только реализация языкового средства, но и соответствующего ему синтаксического сахара. Одной из таких возможностей является авто вывод шаблонных типов, при вызове функций. А наследование классов типов приводит к тому, что кроме поиска реализаций ограничений для класса типа, необходимо также найти соответствующие реализации для классов типов предков. Последовательность действий, а также сам алгоритм вывода шаблонов для классов типов, для обеспечения поддержки синтаксического сахара будут описаны в этой работе.

1.2.3 Инстансы

Для перегрузки классов типов конкретным типами используется языковая конструкция, называемая инстансом. Инстанс определяет то, как конкретный тип реализует класс типов, то есть внутри инстанса переопределяются объявленные в классе типов функции, только теперь шаблонный параметр заменяется на конкретный тип.

Поиск подходящего инстанса происходит во время компиляции. Поэтому нет никакого влияния использования классов типов на время выполнения. В данной работе рассматриваются только случаи с одним инстансом для каждого типа, реализующего определённый класс типа, но перевод кода подразумевает возможность расширения языковой конструкции в этом направлении.

Кроме обычных функций классы типов могут перегружать операторы. Что отражено в примере 1, где для класса типа Group должен быть перегруженный оператор умножения. В случае инстансов, в реализациях всех функций, все операторы интерпретируются, как операторы извне, то есть не принадлежащие классам типов. Подобное поведение необходимо, чтобы избежать рекурсивного зацикливания.

Более подробно алгоритмы выбора подходящего инстанса и перегрузки операторов, также будут рассмотрены в последующих главах.

1.2.4 Функции, ограниченные классами типов

Описанные классами типов ограничения, могут накладываться на параметры шаблонов обобщённых функций. Это означает, что выбранный типовый параметр рассматривается, как инстанс класса типа. При этом все функции, определённые над классом типов, могут быть использованы со всеми переменными имеющими тип соответствующего типового параметра.

В текущей реализации для указания ограничения используется секция where. При этом на один и тот же тип может быть наложено несколько ограничений. В теле функции для обращения к функциям, определённым в классе типов можно использовать явное указание какому классу типов, принадлежит функция, либо если этого не сделать, то функция будет искаться автоматически, но функция сначала ищется в текущем контексте, и если есть функция с таким же именем, то используется именно она. Соответствующие алгоритмы также описаны в последующих главах.

1.3 Классы типов в современных языках программирования

В данной работе мы не рассматриваем методы обобщённого программирования, так как подобное исследование уже проводилось [3]. Важны выводы, сделанные в этом исследовании. На основе анализа, языки программирования были разделены на три категории: функциональные языки программирования, объектно-ориентированные языки и C++. C++ был выделен из-за уникального подхода к обобщённому программированию, называемый шаблонами.

Отличительная особенность шаблонов состоит в том, что это тьюринг-полный язык на этапе компиляции. А значит, что любая конструкция, может быть запрограммирована шаблонами, но шаблоны всё равно имеют ряд недостатков, и часто подвергаются критике [4], в первую очередь за громоздкие и нечитаемые сообщения об ошибках.

Поэтому далее рассматриваются только реализации классов типов как подхода к обобщённому программированию. Как упоминалось ранее, классы типов первоначально были реализованы в функциональных языках

программировании, в частности, в языке Haskell [13]. Но вскоре подобные возможности стали появляться в других языках программирования, например, в Scala[14] и Swift[15].

Стоит отметить, что в отношении многих современных языков программирования, не реализующих подобной конструкции, предпринимаются попытки добавления этой возможности. Ярким примером являются концепты в языке C++ [17]. Либо ведутся дискуссии о необходимости введении подобных конструкций [18].

Глава 2. Перевод классов типов в конструкции языка PascalABC.NET

В данной главе исследуется возможность сведения классов типов к уже имеющимся языковым конструкциям, которые на данный момент используются для реализации обобщённого программирования.

2.1 Перевод наследования класса типов

Эмпирическим путём было выявлено, что наиболее сложная задача, влияющая на генерируемый код, является задача генерации наследования классов типов, поэтому определив, как переводить наследование классов типов мы определим, то как переводить всю языковую конструкцию.

Для решения поставленной задачи было рассмотрено 2 подхода: наследование интерфейсов и дополнительный шаблонный параметр.

2.1.1 Подход наследование интерфейсов

Рассмотрим первый поход «Наследование Интерфейсов». Т.к. язык программирования PascalABC.NET поддерживает только множественное наследование интерфейсов, а в постановке задачи требуется множественное наследование классов типов, то было принято решение рассмотреть перевод классов типов в интерфейсы.

В данном подходе классы типов представляются в виде интерфейсов, а инстансы классов типов, это классы, удовлетворяющие соответствующим ограничениям.

2.1.2 Подход дополнительный шаблонный параметр

Рассмотрим второй подход «Дополнительный Шаблонный Параметр». В этом подходе классы типов переводятся в язык PascalABC.NET в качестве абстрактных классов. А т.к. классы типов представляют собой ограничения на параметры шаблонов, то в качестве наследования этого ограничения можно рассмотреть дополнительный параметр шаблона класса, в который переводится класса типа. Например, если класс типа Ord наследует класс типа Eq, то у класса, в который переводится класс типа Ord, будет дополнительный

шаблонный параметр, который должен быть наследником класса, в который переводится класс типа Eq. Тем самым если возможно инстанцировать этот шаблонный класс, то существует необходимый инстанс класса типа Eq.

2.1.3 Сравнение подхода с наследованием интерфейсов и подхода с дополнительным шаблонным параметром

Сравним эти два подхода. Для начала рассмотрим реализацию по умолчанию. В случае «Наследования Интерфейсов» для реализации поддержки наследования дефолтной реализации классов типов необходимо реализовать дефолтную реализацию интерфейсов. Для «Дополнительного Шаблонного Параметра» это работает по умолчанию, т.к. классы типов транслируются в абстрактные классы.

Важно отметить, что в случае «Наследования Интерфейсов» возникает проблема при которой инстанс явно привязывается к своему предку. Рассмотрим пример:

```
Eq<T> = interface
  function equal(x, y: T): boolean;
  function notEqual(x, y: T) :boolean;
end;
Ord<T> = interface(Eq<T>)
  function compare(x, y: T): Ordering;
  ...
end;
Eq_integer = class(Eq<integer>)
  ...
end;
Ord_integer = class(Eq_integer, Ord<integer>)
public
  function compare(x, y:integer): Ordering;
  begin
    ...
  end;
  ...
end;
```

Из примера видно, что инстанс `Ord_integer` явно привязывается инстансу `Eq_integer`. Следовательно, для другого инстанса `Eq` нужно создавать другой инстанс `Ord`. В случае «Дополнительного Шаблонного Параметра» нет жёсткой привязки благодаря тому, что в качестве наследования используется шаблонный параметр. Рассмотрим пример:

```
Eq<T> = abstract class
```

```
    ...
  end;
  Ord<T, EqT> = abstract class
    where EqT: Eq<T>;

    ...
  end;
  Eq_integer = class(Eq<integer>)
    ...
  end;
  Ord_integer<EqT> = class(Ord<integer, EqT>)
    where EqT: Eq<integer>, constructor;
    ...
  end;
```

Этот пример демонстрирует, что `Ord_integer` параметризован типом, который должен быть наследником `Eq<integer>`, а значит `Ord_integer` не зависит от `Eq_integer`. Рассмотрим функцию, на которую действует ограничение класса типа. При втором подходе, если это ограничение представляет собой класс типа, унаследованный от другого класса типа, то в функции необходимо указать все унаследованные классы типов:

```
function Max3<T, EqT, OrdT>(v1, v2, v3: T): T;
  where EqT: Eq<T>, constructor;
  where OrdT: Ord<T, EqT>, constructor;
```

Но в случае первого подхода для определения функции с классом типом, унаследованным от другого класса типа необходимо указать унаследованный класс типа:

```
function Max3<T, OrdT>(v1, v2, v3: T): T;
  where OrdT: Ord<T>, constructor;
```

Важным преимуществом подхода с наследованием интерфейсов является возможность использования функций базового класса типов с помощью переменной унаследованного класса. Рассмотрим пример:

```
begin
  var ordInst := __ConceptSingleton&<OrdT>.&Instance;
  if ordInst.equal(v1, v2) then
  begin
    ...
  end;
  ...
end;
```

Для вызова метода `equal`, который принадлежит классу типов `Eq` достаточно просто вызвать этот метод у инстанса `Ord`. При использовании

дополнительного шаблонного параметра, необходимо воспользоваться инстансом шаблонного параметра соответствующего класса типа, что значительно усложняет процесс поиска нужной процедуры. Рассмотрим пример:

```
begin
  var eqInst := __ConceptSingleton&<EqT>.&Instance;
  var ordInst := __ConceptSingleton&<OrdT>.&Instance;
  if eqInst.equal(v1, v2) then
  begin
    ...
  end;
end;
```

Для вызова метода `equal`, который принадлежит классу типов `Eq` необходимо использовать переменную инстанса, которая соответствует именно тому ограничению, в котором определен метод, т.е. `eqInst`.

Результаты сравнения двух подходов представлены в *таблице 1*.

	Наследование интерфейсов	Шаблонный параметр
Дефолтная реализация	-	+
Гибкость привязки унаследованных инстансов	-	+
Один шаблонный параметр для унаследованного класса типов	+	-
Использование функций базового класса типов с помощью переменной унаследованного класса	+	-

Таблица 1. Сравнение двух подходов.

2.1.4 Гибридный подход

Как видно из таблицы ни один из подходов не является лучше другого. Чтобы получить подход, обладающий преимуществами над этими двумя подходами, объединим два подхода в один.

Рассмотрим перевод, при котором генерируется обе линейки наследования: и интерфейсы, и абстрактные классы. При этом интерфейсы

будут использоваться для введения ограничений, а абстрактные классы для реализаций по умолчанию.

То есть будут сгенерирован интерфейс, описывающий ограничение класса типа, и абстрактный класс с дефолтными реализациями функций, который реализует этот интерфейс, при этом если класс типа имеет предков, то в класс инстанса и абстрактный класс, соответствующий классу типа, будут иметь количество дополнительных шаблонных параметров, соответствующее количеству непосредственных предков. Кроме того, в эти классы будут добавлены методы из классов типов предков, а реализация делегируется классам типам, определённым шаблонными параметрами.

В свою очередь инстанс будет представлять из себя класс, реализующий интерфейс соответствующего класса типов, и наследующий абстрактный класс этого же класса типов, для использования дефолтных реализаций.

Ограничение на типовые параметры функций переводится в один дополнительный типовый параметр функции, на который накладывается ограничение, в виде соответствующего класса типов.

Проанализируем этот подход по тем же критериям, что и предыдущие подходы:

- Дефолтная реализация – реализуется за счёт абстрактного класса с дефолтными реализациями, в который явно добавляются функции предков, которые делегируют реализацию функциям предкам, заданным шаблонными параметрами.
- Гибкость привязки унаследованных инстансов – достигается за счёт дополнительных параметров шаблонов в классах, генерируемых по инстансам, и абстрактных классах, генерируемых по классам типам. Эти шаблонные параметры соответствуют предкам классов типов, а значит заменяемы.
- Один шаблонный параметр для унаследованного класса типов – выполняется благодаря линейки наследования интерфейсов, то

есть ограничение можно задать одним требованием реализации, соответствующего интерфейса в независимости от количества предков класса типа.

- Использование функций базового класса типов с помощью переменной унаследованного класса – наследование интерфейсов, требует реализации всех функций предков, поэтому унаследованный класс типа должен реализовывать функции базового класса типа.

В итоге, гибридный подход удовлетворяет всем четырём критериям. Единственный недостаток этого подхода – необходимость генерировать и интерфейсы, и абстрактные классы. На данный момент используется этот подход к генерации кода по классам типов. Более подробно, какой код генерируется для каждой конструкции рассмотрено далее.

2.2 Использование атрибутов, для хранения информации о языковой конструкции

Для хранения информации о переведённой конструкции, в данной работе используются атрибуты. Атрибуты — это специальная языковая конструкция, которая позволяет пометить определения некоторым классом, такие классы называются атрибутами. Атрибуты реализованы во многих современных языках программирования, таких как Java, C#, а также атрибуты были добавлены в язык C++ в стандарте C++11.

Такие пометки сообщают компилятору дополнительную информацию о символе. Например, атрибут deprecated есть во многих языках, реализующих предоставляющих подобные языковые возможности. Обычно атрибутом deprecated помечаются символы (подпрограммы, классы, структуры и т.п.), которые устарели, это означает что в последующих версиях программного обеспечения данный символ будет удалён.

Атрибуты зачастую используются для хранения метаинформации, которая необходима для обработки кода или расширения возможностей языка. В текущей реализации атрибуты используются, чтобы сообщать компилятору,

что символ имеет отношения классу типов, кроме того, с помощью атрибутов, передаётся некоторая семантическая информация с уровня синтаксиса на уровень семантики.

Рассмотрим все используемые атрибуты подробнее:

- __TypeclassAttribute – этим атрибутом помечаются все классы типов и инстансы
- __TypeclassInstanceAttribute – этим атрибутом помечаются все инстансы
- __TypeclassMemberAttribute – этим атрибутом помечаются все методы, которые соответствуют определённым в исходном коде функциям, чтобы отличить их от генерируемых функций
- __TypeclassGenericParameter – этим атрибутом помечаются сгенерированные в процессе перевода шаблонные параметры, которые соответствуют классу, реализующему интерфейс, соответствующий классу типов, коме того, этот атрибут принимает на вход один аргумент – строку, который соответствует имени сгенерированной внутри функции переменной, которая хранит инстанс соответствующего класса типа
- __TypeclassRestrictedFunctionAttribute – этим атрибутом помечается функция, в секции where которой есть ограничения классом типом на параметр шаблона

Алгоритмы, отвечающие за разметку кода атрибутами, и то, как эти атрибуты используются при обработке кода, будет рассмотрено в последующих главах.

2.3 Именование генерируемых символов

При замене одного кода другим, возникают новые символы. Например, когда обрабатывается класс типов, необходимо сгенерировать интерфейс, который будет использоваться для ограничения генерируемого шаблонного параметра отвечающего за передачу класса типа.

Для каждого из подобных случаев необходимо использовать единый способ именования. Это не только упростит обработку кода, но и повысит читаемость генерируемого кода.

Рассмотрим все случаи именования генерируемых символов:

- Сгенерированные шаблонные параметры – генерируемый шаблонный параметр соответствует некоторому классу типов и шаблонному параметру, на который накладывается ограничение, сгенерированным именем будет конкатенация имени класса типа и имени шаблонного параметра. То есть, если нужно сгенерировать имя для шаблонного параметра, соответствующего ограничению Ord[TemplateParameter], то результат будет OrdTemplateParameter.
- Класс, соответствующий классу типов – совпадает с именем класса типа, если класс типов определён как Eq[TemplateInstanceType], то результатом будет Eq.
- Интерфейс, соответствующий классу типов – сначала имя генерируется по тем же правилам, что и в предыдущем случае, затем в начале имени добавляется приставка I, то есть для предыдущего примера получается IEq.
- Класс, соответствующий инстансу – сначала генерируется имя класса типа, который реализуется данным инстансом, после чего к этому имени добавляется нижнее подчёркивание, после которого перечисляются типы, для которых этот инстанс определён. Например, инстанс Eq[integer] переводится в Eq_integer.
- Переменная внутри функции, ограниченной классом типов, которая хранит инстанс – к имени сгенерированного шаблонного параметра добавляется окончание Instance. Для примера из первого случая, сгенерируется переменная OrdTemplateParameterInstance.

- Подпрограмма, соответствующая перегрузке оператора – к имени оператора добавляется префикс $typeclass. Если в классе типов переопределён оператор +, то сгенерируется метод с именем $typeclass+.

Далее при описании алгоритмов разворачивания конструкций классов типов в конструкции языка PascalABC.NET считается, что все эти правила выполняются.

2.4 Перевод классов типов

Определим, что необходимо для генерации кода, который заменит класс типов:

- Имя класса типа – TypeclassName. По правилам из раздела 2.3 поставим в соответствие имя интерфейса и обозначим ITypeclassName.
- Шаблонные типы, на которые накладываются ограничения – Parameter1, Parameter2, … ParameterK, где K – количество параметров.
- Предки класса типа – Ancestor1, Ancestor2, … AncestorN, где N – количество предков. Для каждого предка ставится в соответствие имя шаблонного параметра, сгенерированное по правилам из раздела 2.3, обозначим эти имена Ancestor1T, Ancestor2T, … AncestorNT. И пусть IAncestor1, IAncestor2, … IAncestorN – интерфейсы соответстующих классов типов. Пользуясь теми же правилами, определим имена сгенерированных переменных Ancestor1TInstance, Ancestor2TInstance, … AncestorNTInstance.
- Имена функций или процедур с параметрами, определённые в классе типа, возможно с дефолтной реализацией – Method1, Method2, … MethodM, где M – количество методов. Если среди методов есть операторы, то операторы переводятся в соответствии с правилами из раздела 2.3, для простоты

переведённые имена будем обозначать также Method1, Method2, … MethodM.

- Функции классов типов предков – A1Method1, A1Method2, … A2Method1, A2Method2, …

Кроме того, Singleton – специальный класс, для получения синглтона произвольного типа. Теперь рассмотрим перевод класса типа в псевдокод, основанный языке PascalABC.NET, не нарушая общности будем считать, что A1Method1 функция, A1Method2 процедура:

```
type
  ITypeclassName<Pameter1, Parameter2, … ParameterK> =
      interface(IAncestor1<…>, IAncestor2<…>, … IAncestorN<…>)
    Method1;
    Method2;
    …
    MethodM;
  end;

  TypeclassName<Pameter1, Parameter2, … ParameterK,
                Ancestor1T, Ancestor2T, … AncestorNT> =
    abstract class(ITypeclassName<Pameter1, Parameter2, …
                                  ParameterK>)
  where Ancestor1T: IAncestor1<…>, constructor;
  where Ancestor2T: IAncestor2<…>, constructor;
  …
  where AncestorNT: IAncestorN<…>, constructor;

  public
    Method1; virtual;
    Method2; virtual;
    …
    MethodM; virtual;

    A1Method1; virtual;
    begin
      var Ancestor1TInstance := Singleton<Ancestor1T>.Instance;
      Result := Ancestor1TInstance.A1Method1(…);
    end;
    A1Method2; virtual;
    begin
      var Ancestor1TInstance := Singleton<Ancestor1T>.Instance;
      Ancestor1TInstance.A1Method1(…);
    end;
    …
  end;
```

Листинг 1

Из листинга 1 видно, что каждый класс типа переводится в интерфейс и абстрактный класс, при это предки в интерфейсе просто наследуются, а в классе типов переводятся в параметр шаблона, на который накладывается ограничение, соответствующего интерфейса. Также в абстрактный класс добавляются методы классов предков с делегированием реализации соответствующему предку. Кроме того, код размечается атрибутами, исходя из правил, описанных в разделе 2.2.

2.5 Перевод инстансов

Определим информацию о инстансе, которую нужно извлечь, чтобы сгенерировать код инстанса:

- Имя класса типа – TypeclassName. По правилам из раздела 2.3 поставим в соответствие имя интерфейса и обозначим ITypeclassName. По тем же правилам генерируется имя инстанса TypeclassName_Type1Type2…TypeK
- Конкретные типы, на которые накладываются ограничения – Type1, Type2, … TypeK, где K – количество параметров
- Предки класса типа – Ancestor1, Ancestor2, … AncestorN, где N – количество предков. Аналогично предыдущему разделу, также определяются: Ancestor1T, Ancestor2T, … AncestorNT и IAncestor1, IAncestor2, … IAncestorN.
- Имена функций или процедур с параметрами, определённые в инстансе, с реализацией – Method1, Method2, … MethodM, где M – количество методов. Методы переименовываются аналогично предыдущему разделу. Предполагается, что есть реализация для всех методов. Она либо определена в инстансе, либо определена дефолтная реализация. Если реализация определена и там, и там, то приоритет отдаётся реализации из инстанса

Рассмотрим перевод инстанса в псевдокод:

```
type
  TypeclassName_Type1Type2…TypeK<Ancestor1T,Ancestor2T, …
      AncestorNT> = class(TypeclassName<Type1, Type2, … TypeK,
                          Ancestor1T, Ancestor2T, … AncestorNT>,
                          ITypeclassName<Type1, Type2, … TypeK>)
  where Ancestor1T: IAncestor1<…>, constructor;
  where Ancestor2T: IAncestor2<…>, constructor;
  …
  where AncestorNT: IAncestorN<…>, constructor;

  public
    Method1; override;
    Method2; override;
    …
    MethodM; override;
  end;
```

Листинг 2

Из листинга 2 можно сделать вывод, что инстанс принимает предков, как параметр шаблона, каждый из этих параметр реализует интерфейс класса типа предка. И хотя явно в реализации эти параметры шаблона не используются, они передаются абстрактному классу этого класса типа, который на основе этих классов, строит методы из классов типов предков. А значит это делается для каждого класса типа только один раз – в момент генерации класса типа. Методы помечаются override, и все символы размечаются атрибутами в соответствии с разделом 2.2.

2.6 Перевод обобщённых подпрограмм, ограниченных классом типов

Рассмотрим информацию, которую нужно собрать о функции, которая определена с ограничением класса типа на типовый параметр:

- Имя подпрограммы, параметры, ключевое слово (функция или процедура) – остаются неизменными. Не нарушая общности будем считать, что переводится функция с именем RestrictedFunction, параметры обозначим (…) и возвращаемое значение ReturnType.
- Типовые параметры обозначим TemplateType1, TemplateType2, … TemplateTypeN

- Все ограничения классов типов вида: where Typeclass1[C1], где C1 – любое сочетание с повторениями типовых параметров. Важно отметить, что среди Typeclass1, Typeclass2, … TypeclassM могут быть одинаковые имена, но в текущем контексте мы их обозначаем разными алиасами. Согласно правилам именования из раздела 2.3 получим имена соответствующих интерфейсов: ITypeclass1, … и имена шаблонных параметров Typeclass1C1, ... А также для каждого такого случая по тем же правилам определим имена переменных, хранящих инстансы и обозначим эти имена следующим образом: Typeclass1C1Instance, Typeclass2C2Instance, … TypeclassMCMInstance.

Рассмотрим перевод подобной функции в псевдокод, в этой части кода также используется Singleton:

```
function RestrictedFunction<
           TemplateType1, … TemplateTypeN,
           Typeclass1C1, … TypeclassMCM>(…): ReturnType;
  where Typeclass1C1: Typeclass1[C1];
  …
  where TypeclassMCM: TypeclassM[CM];
begin
  var Typeclass1C1Instance := Singleton<Typeclass1C1>.Instance;
  …
  var TypeclassMCMInstance := Singleton<TypeclassMCM>.Instance;

  …
end;
```

Листинг 3

Листинг 3 демонстрирует, что все ограничения переводятся в шаблонный параметр, ограниченный интерфейсом класса типа. Кроме того, для каждого класса типа добавляется в тело подпрограммы переменные, хранящие инстанс. И в последующем коде все вызовы функций из классов типов необходимо производить с использованием соответствующей переменной. Эта часть в псевдокоде не отражена, но будет подробно рассмотрена при описании алгоритма перевода.

Глава 3. Синтаксис классов типов в языке PascalABC.NET

В данном разделе рассматривается представление языковой конструкции классов типов в языке программирования PascalABC.NET на уровне грамматики и синтаксического дерева. Эти два этапа реализации языковой конструкции тесно взаимосвязаны и определяют то, как программист может использовать данную конструкцию.

Для реализации грамматики в языке программирования PascalABC.NET используется генератор сканеров GPLex[6] и генератор парсеров GPPG[7]. Этот набор утилит используется для написания LALR(1) грамматик[8] на языке программирования C#. Грамматика задаётся в виде правил похожих на форму Бэкуса – Наура. При описании грамматики конструкции, будет указано, в каком месте грамматики была встроена данная конструкция, а также как именно конструкция определена в БНФ.

Модификации синтаксического дерева, а также модификации всего связанного с деревом кода, то есть визиторов, в языке PascalABC.NET производится автоматизировано, с использованием утилиты NodesGenerator, которая позволяет хранить информацию о узлах в специальном XML формате и генерирует по этой информации классы, которые являются узлами синтаксического дерева, и визиторы по этим узлам.

```
<SyntaxNode Name="ident" BaseName="addressed_value_funcname">
  <Fields>
     <ExtendedField Name="name" Type="string"
CreateVariable="false" DeleteVariable="false" />
  </Fields>
  <Methods />
  …
</SyntaxNode>
```

Пример 4

В примере 4 рассматривается фрагмент xml кода, соответствующего узлу ident, который соответствует именованному идентификатору. И можно заметить, что у этого узле есть поле name, имеющее тип string. Информация о тэгах, которые используются для категоризации узлов, пропущена для краткости.

3.1 Синтаксис классов типов

Рассмотрим определение класса типа на псевдокоде близком к языку PascaABC.NET:

```
type
  TypeclassName[TemplateParam1, …] = typeclass(
    BaseTypeclass1[TypeReference1, …], …)

   function operator+(v1, v2: TemplateParam1): TemplateParam1;

   …

   function Method1(input: InputType): ReturnType;
   begin
     ProcedureFromBaseTypeclass1();
     …
   end;

   function Method2(input: InputType): ReturnType;

   …

  end;
```

Листинг 4

Из листинга 1 видно, что в классе типов могут быть определены операторы над типами, ни один из которых не является классом типов. Также Method1 демонстрирует реализацию подпрограммы с дефолтной реализацией, внутри которой вызывается функция, принадлежащая классу типу предку. А предки указываются в скобках после ключевого слова typeclass, которое говорит компилятору, что определяемый символ — это класс типа.

Выразим эти правила в БНФ:

```
<простое определение типа> ::=
    <определение типов до добавления классов типов>
  | <ограничение класса типа> = typeclass<опц. список предков>
    <опц. список компонентов типа, заканчивающийся end> ;

<ограничение класса типа> ::=
  <простой идентификатор типа> [ <список параметров типов> ]
```

Здесь опц. – это сокращение для опциональный, т.е. нетерминал может быть пустым. В процессе разбора этого выражения получается специальный синтаксический узел.

Для работы с этой конструкцией был добавлен узел typeclass_definition, который создаётся на основе правил, описанных выше. Он является наследником type_definition, и имеет поля:

- additional_restrictions – поле типа named_type_reference_list, хранит список предков класса типа
- body – имеет тип class_body_list, хранит всё, что определяется внутри класса типа

Для ограничения класса типа используется узел typeclass_restriction – наследник ident, имеющий поле resctriction_args типа template_param_list, хранящее список типов, на который накладывается ограничение. А имя ограничения хранится в поле name унаследованном от ident.

На основе этих двух узлов строится type_declaration, который далее обрабатывается стандартными методами.

3.2 Синтаксис инстансов

Рассмотрим, как можно определить инстанс класса типа на псевдокоде:

```
type
  TypeclassName[ConcreteType1, …] = instance

    procedure Method1(value: ValueType1; …);
    begin
      var tmp := FunctionFromTypeclassAncestor(value);
      …
    end;

    function Method2(input: InputType): ReturnType;
    begin
      …
    end;
    …
  end;
```

Листинг 5

В листинге 5 показано, что синтаксис определения инстанса очень похож на синтаксис определения класса типа. Отличие состоит в ключевом слове, instance вместо typeclass, а также в отсутствии предков, т.к. в инстансе просто перегружаются функции в уже имеющейся иерархии. Кроме того, хотя в инстансе явно не указываются предки, функции, определённые в классах

типах предках можно использовать внутри реализаций подпрограмм, как это продемонстрировано в Method1.

Выразим эти правила в БНФ:

```
<простое определение типа> ::=
    <определение типов до добавления классов типов>
  | <определение класса типа>
  | <ограничение класса типа> = instance
    <опц. список компонентов типа, заканчивающийся end> ;
```

Для определения инстанса используется аналогичный определению классу типов синтаксический узел instance_definition – наследник type_definition, имеющий поле body типа class_body_list, хранящее информацию, о предоставленных в инстансе реализациях.

В итоге, на основе typeclass_restriction и typeclass_definition также строится type_declaration.

3.3 Синтаксис ограничений на обобщённые подпрограммы

Рассмотрим определение ограничения на обобщённую подпрограмму на псевдокоде:

```
function GenericFunctonName<Template1, …>(param1: ParamType1;…):
                                                        ReturnType;
  where TypeclassName1[C1];
  …
begin
  var tmp := TypeclassName1&[C1].ExplicitTypeclassFunction(…);
  ImplicitProcedure(…);
  var template1X, template1Y: Template1;
  …
  var operatorOverloadingResult := template1X * template1Y;
end;
```

Листинг 6

В листинге 6 приведён синтаксис только для ограничения обобщённой функции, но для обобщённой процедуры синтаксис аналогичный с тривиальными заменами. В этом коде C1, C2, … – комбинации некоторых типов, среди которых могут быть и шаблонные типы Template1, Template2, … В строке определением переменной tmp, демонстрируется явное использование функции, определённой в классе типов. Кроме того, листинг демонстрирует не явное использование функции и вызов перегруженного оператора умножения для типа Template1.

Опишем, используя БНФ, то как эта конструкция повлияла на грамматику языка PascalABC.NET:

```
<элемент секции where> ::=
    <элмент секции where до внедрения классов типов>
  | where <ограничение класса типа> ;
```

Новому элементу секции where соответствует синтаксический узел where_typeclass_constraint, наследуемый от узла where_definition и содержащий поле restriction класса typeclass_restriction, отвечающее за хранение ограничения. Все полученные секции where для конкретной функции добавляются в соответствующий узел where_definition_list.

Глава 4. Алгоритмы перевода классов типов в язык PascalABC.NET

Обработка исходного текста программ на языке PascalABC.NET, как и в любом современном языке программирования, – сложный и многоступенчатый процесс. В предыдущих главах уже рассматривался этап парсинга, в результате которого получается синтаксическое дерево программы, так же ещё можно выделить этапы:

- синтаксический – этап на котором синтаксическое дерево переводится в синтаксическое дерево, но зачастую множество уникальных узлов программы уменьшается, т.к. происходит разворачивание синтаксического сахара
- семантический – этап перевода синтаксического дерева в семантическое [9], отличительной особенностью этого этапа является построение специальных структур, хранящих семантическую информацию о программе, таких как таблица символов [10]
- генерация кода – этап перевода семантического дерева в IL-код

На синтаксическом этапе в проекте язык программирования PascalABC.NET реализован широкий спектр различных инструментов [12]. Одним из таких инструментов является рассмотренный ранее NodesGenerator. Обработка дерева производится с помощью паттерна визитор [11], который обеспечивает необходимую двойную диспетчеризацию.

Для разных целей реализованы разные линейки визиторов:

- WalkingVisitorNew – визитор с определённым дефолтным обходом для всех узлов
- BaseEnterExitVisitor – визитор, которому можно задать специальные действия на пред и пост обработку узлов
- CollectUpperNodesVisitor – визитор, хранящий путь от корня до текущего узла в виде стека

- BaseChangeVisitor – визитор для замены одних узлов другими, используется для разворачивания синтаксического сахара

На этих визиторах основана вся обработка представления программы на синтаксическом этапе.

На семантическом этапе собирается и выводится вся информация о типах, поэтому этот этап тесно связан со сложными алгоритмами вывода типов и, в частности, вывода типа шаблонов, для решения этих задач на этапе семантики предоставляется набор инструментов:

- SymbolTable.Scope – таблица символов, построенная на связанных между собой скоупах, используется для запросов к уже обработанным символам
- compilation_context – информация о текущем состоянии, например, текущий скоуп, последняя обработанная функция и т.п.
- conversion_data_and_algorithms – набор алгоритмов для построения семантического дерева, вывода и преобразования типов, и т.п.
- generic_conversions – алгоритмы необходимые для работы с обобщёнными данными и алгоритмами

То, как эти инструменты будут использоваться в реализации языковой конструкции, будет видно из описания последующих алгоритмов.

4.1 Реализация паттерна «одиночка» для инстансов

В главе 2, при описании перевода классов типов и обобщённых подпрограмм возникал класс, который отвечал за получение единственного экземпляра определённого типа.

Чтобы избежать генерирования каждый раз этого класса, был использован подход, при котором код необходимого элемента добавляется в системную библиотеку, которая не явно подключается ко всем программам. В случае PascalABC.NET, это PABCSystem.pas.

В указанную библиотеку был добавлен класс __ConceptSingleton<T>, при этом на тип T накладывается ограничение constructor, т.е. T – только такие типы, у которых есть конструктор без параметров. Это необходимо, чтобы создавать объект в случае необходимости. У этого класса есть классовый метод Instance, который либо возвращает уже созданный инстанс, либо создаёт новый, запоминает его и возвращает только созданный элемент.

Стоит отметить, что подобный подход гарантирует, что всегда будет только один инстанс, в случае, когда к типам, которые должны быть одиночками обращаются только через этот класс. Учитывая, что этот вспомогательный класс будет использоваться только при генерации, мы можем гарантировать выполнение этого условия.

4.2 Алгоритмы на синтаксическом уровне

За частую часть преобразований при реализации синтаксического сахара можно выполнить переводом синтаксического дерева в синтаксическое.

У этого подхода есть ряд преимуществ, в частности, нет необходимости перестраивать таблицу символов, при замене одной структуры на другую, но при этом информации о символах для некоторых случаев может не хватать.

Поэтому прежде, чем производить обработку языковой конструкции на этом уровне, необходимо собрать информацию об обрабатываемых символах. Для этих целей в данной работе используется два визитора:

- FindTypeclassesVisitor – первый проход по синтаксическому дереву с классами типов, за этот проход собираются классы типов в виде словаря, ключом которого является имя класса типа, а значением – узел type_declaration, в котором определяется класс типа с таким именем.
- FindInstancesAndRestrictedFunctionsVisitor – вторым проходом, собираются инстансы и обобщённые подпрограммы, которые ставятся в соответствие классам типам, то есть для каждого

класса типа хранится список его инстансов и список его подпрограмм.

- ReplaceTypeclassVisitor – визитор, который основываясь на информации, полученной на предыдущих этапах, генерирует реализацию, которая заменяет классы типов в исходном коде

Первые два визиторы выполняют только сбор информации о символах, на этих этапах нет необходимости производить замену одного кода на другой. Именно поэтому они являются наследником класса WalkingVisitorNew, описанного ранее. Реализация этих визиторов тривиальна.

Далее будет рассмотрен только последний этап. Цель этого этапа выполнить перевод исходного кода с классами типов, имеющего вид описанный в главе 3, в код описанный в главе 2, в соответствии с правилами, также описанными в главе 2.

4.2.1 Алгоритм перевода классов типов

В соответствии с разделом 2.4 необходимо сгенерировать интерфейс, соответствующий классу типу, и абстрактный класс с реализацией по умолчанию и реализацией функций предков. Также полученный в результате код должен быть размечен атрибутами по правилам раздела 2.2.

Рассмотрим подробно шаги алгоритма. Для начала из синтаксического узла необходимо извлечь информацию необходимую для генерации кода:

- Имя класса типа – используется для определении имён сгенерированных структур
- Набор определений подпрограмм с возможной реализацией по умолчанию, формирующих класс типа

На основе полученных данных необходимо сгенерировать код для интерфейса, соответствующего этому классу типов:

- Перевести подпрограммы в методы интерфейса, для этого выполнить для каждого метода следующее:
  - ♦ Если подпрограмма определена только как заголовок, то оставить её как есть

- Если подпрограмма определена с реализацией по умолчанию, то опустить реализацию и преобразовать в заголовок
- Добавить к полученному атрибут __TypeclassMemberAttribute
- Если имя заголовка это operator_name_ident, то заменить имя на обычное с префиксом «$typeclass»

- Если у класса типов есть предки, то получить имя интерфейса каждого класса типа предка и унаследовать генерируемый интерфейс от полученных интерфейсов
- Пометить генерируемый интерфейс атрибутом __TypeclassAttribute

После того, как сгенерирован интерфейс необходимо сгенерировать абстрактный класс:

- Перевести подпрограммы в методы абстрактного класса, для этого выполнить для каждого метода следующее:
  - Если подпрограмма определена только как заголовок, то оставить её как есть, и пометить её abstract
  - Если подпрограмма определена с реализацией по умолчанию, то оставить её как есть, и пометить её virtual
  - Добавить к полученному атрибут __TypeclassMemberAttribute
  - Если имя заголовка это operator_name_ident, то заменить имя на обычное с префиксом «$typeclass»
- Для каждого предка текущего класса типа выполнить:
  - Сгенерировать по предку шаблонный параметр и добавить его к шаблонным параметрам генерируемого класса
  - Добавить соответствующую секцию where, для шаблонного параметра, сгенерированного на предыдущем шаге, потребовать, чтобы он реализовывал интерфейс класса типа предка и имел конструктор по умолчанию

- ♦ Добавить методы из предка, для каждого метода необходимо выполнить:
  - ➢ Добавить в генерируемый класс заголовок метода
  - ➢ Добавить в реализацию метода из предыдущего шага получение инстанса предка и сохранение в локальную переменную с помощью класса описанного в 4.1
  - ➢ Если этот метод функция, то присвоить переменной Result результат выполнения метода предка, вызвав его используя инстанс с предыдущего шага и параметры сгенерированного метода
- Пометить генерируемый абстрактный класс атрибутом __TypeclassAttribute

После выполнения этих шагов для конструкции, описанной в разделе 3.1, получится код из раздела 2.4, размеченный атрибутами в соответствии с правилами 2.2

4.2.2 Алгоритм перевода инстансов

В случае инстанса, необходимо сгенерировать класс, реализующий интерфейс соответствующего класса типа с шаблонными параметрами интерфейса, соответствующим параметрам ограничений класса типа.

Для того, чтобы сгенерировать код для инстанса, необходимо из исходного кода извлечь следующую информацию:

- Имя класса типа – необходимо, чтобы определить к какому именно классу типа принадлежит этот инстанс, а также для определения имён сгенерированных символов
- Классы типов предки класса типа определённого в предыдущем шаге
- Конкретные типы, для которых определятся инстанс
- Набор определений подпрограмм, определяющих то, как инстанс реализует класс типа

Рассмотрим подробно шаги алгоритма генерации кода на основе полученных данных:

- Для каждого предка класса типа:
  - ♦ Сгенерировать шаблонный параметр по имени предка
  - ♦ Добавить шаблонный параметр из предыдущего шага к шаблонным параметрам генерируемого класса
  - ♦ Добавить соответствующую секцию where, для шаблонного параметра, сгенерированного на предыдущем шаге, потребовать, чтобы он реализовывал интерфейс класса типа предка и имел конструктор по умолчанию
- Добавить предков для генерируемого класса:
  - ♦ абстрактный класс соответствующего класса типов с шаблонными параметрами, представляющими собой конкатенацию конкретных типов и обобщённых типов, соответствующих предкам
  - ♦ интерфейс соответствующего класса типов с шаблонными параметрами, соответствующим конкретным типам
- Добавить в генерируемый класс методы из подпрограмм инстансов, пометив их как override, и переименовав операторы, если таковые имеются
- Пометить генерируемый класс атрибутами __TypeclassAttribute и __TypeclassInstanceAttribute

Эта последовательность шагов переводит конструкцию из раздела 3.2 в код из раздела 2.4.

4.2.3 Алгоритм перевода определения обобщённых подпрограмм

При переводе определения обобщённой подпрограммы с ограничениями на параметры шаблона, получается другая обобщённая функция с большим числом параметров шаблонов.

Определим информацию, которую необходимо получить из исходного кода для генерации перевода:

- Имя функции
- Тело функции
- Секция where
- Исходные шаблонные параметры

Далее подробно рассматривается алгоритм, который на основе собранной информации генерирует новую функцию:

- В секции where для каждого ограничения являющимся where_typeclass_contraint необходимо выполнить:
  - ♦ Сгенерировать параметр шаблона генерируемой функции, соответствующий классу типов из ограничения
  - ♦ Заменить это ограничение требованием, чтобы параметр шаблона из предыдущего шага реализовывал интерфейс соответствующего класса типа из ограничения
  - ♦ Сгенерировать имя для переменной, хранящей инстанс, типа совпадающего с новым параметром шаблона
  - ♦ В начале блока добавить определение сгенерированной на предыдущем шаге переменной, инициализированной синглтоном, соответствующего типа
  - ♦ Пометить определение этого шаблонного параметра атрибутом __TypeclassGenericParameter, на вход этому атрибуту подать имя сгенерированной ранее локальной переменной
- Заменить все явные указания класса типа в теле генерируемой функции на соответствующие локальные переменные, сгенерированные на предыдущем шаге
- Пометить генерируемую функцию атрибутом __TypeclassRestrictedFunctionAttribute

Эта последовательность шагов переводит конструкцию из раздела 3.3 в код из раздела 2.5.

4.3 Перевод на уровне семантики

Уровень семантики для разворачивания языковой конструкции в базовый язык используется в случаях, когда для перевода необходима семантическая информация о символах, за частую связанная с выводом типов.

Формально рассматриваемой конструкции на семантическом этапе уже нет, т.к. все синтаксические узлы, связанные с данной конструкцией, были переведены в синтаксические узлы базового языка.

Но тем не менее за счёт передачи дополнительной информации о символах с помощью разметки их атрибутами, можно улучшить алгоритмы, связанные с выводом типов, используя дополнительное знание о символах, как о коде, сгенерированном при переводе конструкции классов типов.

Например, используя атрибуты мы всегда можем сказать имеет ли текущий обрабатываемый участок кода отношение к классам типам. Этот подход используется в большинстве описанных далее алгоритмах.

Используя compilation_context, всегда можно сказать, если в данный момент обрабатывается подпрограмма или тип, а также получить информацию о соответствующем символе. Если этот символ размечен одним из атрибутов имеющим отношение к классу типов, то это означает, что обрабатываемый код связан с классом типом, и в необходимых случаях становится возможно определить специальное поведение, связанное с классами типов.

Далее в случае необходимости будет просто описано условие для этого специального поведения без указания того, как эта информация была получена.

4.3.1 Перевод операторов

На синтаксическом уровне операторы, определённые в классах типов, уже переводятся в функции с особенным именем, осталось модифицировать алгоритм поиска оператора.

На семантическом уровне для поиска оператора используется вспомогательный алгоритм find_operator.

Прежде чем мы перейдём к реализации, стоит отметить, что в функции и классе типов логично использовать операторы, определённые в классе типов, в случае же инстансов, могут понадобиться операторы из внешнего контекста, для переопределения оператора класса типа текущим, имеющимся определением оператора. Поэтому для инстансов поиск операторов в классах типов производится не будет. Чтобы избежать рекурсивного зацикливания.

Рассмотрим модифицированную часть алгоритма:

- Если текущая обрабатываемая функция – обобщённая функция с ограничением параметра шаблона
  - ♦ Выбрать из обобщённых параметров функции – параметры, соответствующие классам типов
  - ♦ Найти в этих классах типах походящий метод, имеющий имя $typeclass + имя оператора
  - ♦ Если такой метод есть, то используя аргумент атрибута __TypeclassGenericParameterAttribute, который определяет имя локальной переменной инстанса, заменить использование оператора вызовом этого метода
- Если текущий обрабатываемый тип имеет отношение к классу типов и не имеет отношения к инстансам
  - ♦ Если интерфейс, который данный класс реализует имеет метод с именем $typeclass + имя оператора
    - ➢ Преобразовать использование оператора в вызов метода с этим специальным именем через переменную self

Этот алгоритм позволяет использовать операторы как внутри подпрограмм с ограничением классами типов на параметры шаблона, так и при определении реализаций по умолчанию в классах типов.

4.3.2 Вывод параметров шаблона для инстансов классов типов, которые унаследованы от других классов типов

Это вспомогательный алгоритм, который нужен при выводе параметров шаблонов обобщённых подпрограмм, ограниченных классами типов.

Когда ищется инстанс подходящего класса типа, нужно так же убедится, что все предки этого класса типа так же имеют соответствующие инстансы. Так как важны все предки, а не только непосредственные, то в результате получается рекурсивный алгоритм.

Рассмотрим подробно шаги этого алгоритма:

- Для каждого типа, представляющего класс типа выполнить
  - ♦ Найти подходящий инстанс
  - ♦ Если этот инстанс реализует класс типа без наследников, то
    - ➢ Вывод этого шаблона закончен
  - ♦ Иначе
    - ➢ Рекурсивный вызов алгоритма для вывода инстансов предков классов типов

В результате работы этого алгоритма получается полностью выведенный инстанс класса типа, который имеет предков, если определены все необходимые предкам инстансы.

4.3.3 Вывод параметров шаблонов обобщённых подпрограмм, ограниченных классами типов

Алгоритм вывода параметров шаблонов – многоступенчатый, итеративный алгоритм, в процессе которого, если возможно по типам фактических входных параметров выводятся типы шаблонов обобщённой подпрограммы.

Если подпрограмма имеет отношения к классам типов, значит у неё есть шаблонный параметр, используемый для передачи инстанса этой обобщённой подпрограмме. И именно эти параметры шаблонов необходимо выводить особенным способом.

Рассмотрим часть итеративного алгоритма вывода типов шаблонов, посвящённую классам типов:

- Для всех шаблонных параметров, помеченных атрибутом __TypeclassGenericParameter выполнить:
  - ♦ Найти подходящий инстанс, если не удалось найти инстанс записать в deduced с соответствующим индексом null
  - ♦ Если этот инстанс реализует класс типа без наследников, то
    - ➢ Вывод этого шаблона закончен, записать выведенный тип в deduced с соответствующим индексом
  - ♦ Иначе
    - ➢ Вызвать вспомогательную функцию для вывода рекурсивных инстансов, в случае успеха результат записать в deduced с соответствующим индексом, иначе записать null

Это языковое средство позволяет опускать шаблонные параметры при вызове функции, и если параметры шаблона можно вывести по типам фактических параметров и для этих шаблонов есть необходимые инстансы, то эти подстановки будут выполнены автоматически.

4.3.4 Поиск функций в классах типах, в случае, когда явно не указано, что функция принадлежит к классу типов

Эта возможность позволяет опускать указание из какого класса типа функция. И в случае отсутствия неоднозначности подходящая функция будет выбрана, основываясь на контексте, в котором был вызов.

Для введения этого расширения необходимо встроится в visit_method_call. При поиске метода в случае, когда имя метода задано идентификатором, если ничего не было найдено, то нужно попробовать найти метод в используемых в данном контексте классах типов.

Рассмотрим более подробно алгоритм поиска метода в классах типах:

- Если обычным способом символ не был найден и в текущий момент обрабатывается реализация обобщённой функции с ограничением классами типов на параметры шаблонов

- ♦ Получить все шаблонные параметры, соответствующие классам типам
- ♦ Получить для этих шаблонных имена локальных переменных
- ♦ Если в типах, соответствующих классам типов находится подходящий метод, то необходимо заменить вызов функции вызовом метода экземпляра класса, заданного локальной переменной

- Обработать как случай dot_node

Эта языковая конструкция упрощает пользовательский код и делает его менее многословным.

Заключение

В диссертационной работе проведены исследования по реализации классов типов языке PascalABC.NET, в итоге был выбран оптимальный подход для реализации. В процессе работы были решены следующие задачи:

1. Реализованы классы типов для языка PascalABC.NET с поддержкой следующих возможностей:
   - Множественное наследование классов типов
   - Реализация по умолчанию для подпрограмм определённых в классе типов
2. Обеспечена поддержка перегрузки операторов
   - Переопределение операторов внутри класса типа для элементов произвольного типа
   - Поиск операторов в классе типов, если таковых нет во внешнем контексте
3. Обеспечен поиск подпрограммы в классе типов, если явно не указано, что подпрограмма принадлежит классу типу, и при этом нет подходящей подпрограммы во внешнем контексте
4. Модифицирован алгоритм вывода шаблонных параметров для поиска необходимого инстанса, при инстанцировании обобщённых подпрограмм на которых наложено ограничение классами типов.

Классы типов – гибкий и удобный инструмент для использования парадигмы обобщённого программирования. И хотя эта языковая конструкция возникла в функциональном языке программирования, её аналоги нашли применение в языках Scala и Swift, также концепты для C++ уже долгое время являются предметом частых дискуссий. Классы типов набирают популярность за счёт своей немногословности по сравнению с дженериками и простотой сообщений об ошибках по сравнению с шаблонами.

Список литературы

1. Сайт проекта PascalABC.NET – http://pascalabc.net
2. Ткачук А.В. Язык, компилятор и система программирования PascalABC.NET. Дипломная работа, Южный Федеральный Университет, 2007.
3. Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock, An Extended Comparative Study of Language Support for Generic Programming, 2007
4. Пеленицын А.М., Концепты C++17 в их отношении к концептам C++0X, Современные информационные технологии и ИТ-образование, 2015
5. Иванов С.О. Языковые средства и генерация кода шаблонов классов для PascalABC.NET. Магистерская диссертация, Южный Федеральный Университет, 2008.
6. Сайт проекта Garden Points LEX (gplex) – https://archive.codeplex.com/?p=gplex
7. Сайт проекта Garden Points Parser Generator (gppg) – https://archive.codeplex.com/?p=gppg
8. Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, Second Edition, 2006
9. Водолазов Н.Н. Конвертор в семантическое дерево для компилятора PascalABC.NET. Магистерская диссертация, 2007
10. Ткачук А.В. Таблица символов и генерация синтаксического дерева для компилятора PascalABC.NET. Курсовая работа, 2006
11. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1994
12. Захаренко С.А. Автоматическая генерация кода для узлов синтаксического дерева и визиторы на изменение синтаксического дерева, бакалаврская квалификационная работа, 2016

13. Сайт проекта Haskell – https://www.haskell.org/
14. Сайт проекта Scala – https://www.scala-lang.org/
15. About Swift — The Swift Programming Language (Swift 4.2). — URL: https://docs.swift.org/swift-book/index.html
16. Джозеф Албахари Б. А. C# 5.0. Справочник. Полное описание языка. — 2013.
17. B. Stroustrup and A. Sutton, A Concept Design for the STL, 2012
18. Julia Belyakova, Stanislav Mikhalkovich. Pitfalls of C# Generics and Their Solution Using Concepts, 2015
19. Система контроля версий GitHub. — URL: https://github.com/