МИНОБРНАУКИ РОССИИ

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

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

Кафедра информатики и вычислительного эксперимента

Калинин Александр Александрович

Реализация конструкций async и await для языка программирования PascalABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

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

Допущено к защите:
руководитель направления _____ Михалкович С.С.

Ростов-на-Дону – 2024

Оглавление

Введение

Программирование, где результат доступен не сразу, а после некоторого времени через асинхронный вызов, представляет собой концепцию асинхронного процесса. В отличие от синхронного программирования, где в результате последовательного выполнения инструкций происходит блокировка операций, асинхронный подход позволяет запускать длительные операции без задержек и блокировки следующего этапа выполнения программы.

В современных высокоуровневых языках программирования, таких как C# и JavaScript, для того чтобы можно было работать с асинхронными вызовами, применяются ключевые слова: **async** и **await**, которые используются для упрощения написания асинхронных методов. При этом асинхронный код с применением async и await выглядит как синхронный код.

В данной работе описывается процесс проектирования и реализации данных конструкций для языка программирования PascalABC.NET, а также их интеграция в существующую инфраструктуру проекта.

1. Постановка задачи

Для реализации конструкций async и await для языка программирования PascalABC.NET требовалось выполнить следующие задачи:

1) Изучить, как реализованы async и await на других языках высокого уровня (C#, JavaScript)
2) Дополнить грамматику языка PascalABC.NET конструкциями async и await.
3) Добавить новые узлы в синтаксическое дерево и новые поля к существующим узлам
4) Реализовать конечный автомат для создания машины состояний асинхронного метода
5) Добавить развертку(lowering) управляющих конструкций языка для работы асинхронного кода внутри циклов, условий, и т. д.
6) Реализовать алгоритм работы с локальными и глобальными переменными
7) Предусмотреть возможность работы внутри классов, а также со статическими методами и переменными

2. Исследование предметной области

Перед тем, как приступить к выполнению поставленных задач, мне нужно было разобраться с тем, что такое асинхронность и для каких целей применяются конструкции async и await. Асинхронность предоставляет возможность выполнения каких-либо задач, не дожидаясь сигнала об их завершении и не блокируя основной поток выполнения.

Так, например, если есть задачи, требующие времени на загрузку данных из сети или выполнение каких-либо продолжительных операций, то применение асинхронности с конструкциями async и await позволяет выпонлять такие операции эффективно и без блокировки основного потока выполнения.

После того, как я разобрался с понятием асинхронности, мне нужно было изучить, как реализованы async и await на других языках высокого уровня. Самой подходящей оказалась реализация async и await для языка C#.

На языке C#, чтобы указать, что метод стал асинхронным требуется пометить его с помощью слова async. Кроме того, возвращаемый тип данного метода должен быть **Task<T>**:

Листинг 1. Пример простейшей асинхронной программы на языке C#

```
static async Task Main()
 {
     Console.WriteLine("MainStart");
     await Task.Delay(2000);
     Console.WriteLine("MainEnd");
 }
```

При этом внутренняя реализация данного асинхронного метода выглядит так:

Листинг 2. Внутренняя реализация асинхронного метода Main

```
     private static Task Main()
     {
          <Main>d__0 <Main>d__ = new <Main>d__0();
          <Main>d__.<>t__builder =
AsyncTaskMethodBuilder.Create();
          <Main>d__.<>1__state = -1;
          <Main>d__.<>t__builder.Start(ref <Main>d__);
```

```
        return <Main>d__.<>t__builder.Task;
    }
```

Следует отметить, что конструкция async указывает на то, является метод асинхронным или же нет. В то же время конструкция await не позволяет выполняться коду, находящемуся ниже await до тех пор, пока асинхронный метод не закончит свою работу. Это своеобразная контрольная точка. При этом асинхронный метод типа **Task<T>** может закончить свою работу с исключением или вернуть некий результат типа **<T>**.

Кроме того, ждать завершение асинхронного метода вовсе не обязательно. Если результат работы метода нам не важен, то можно обойтись и без конструкции await.

Так как await не допускает блокировку потока при длительной работе асинхронного метода, то этот поток возвращается в пул потоков, откуда его потом можно будет повторно использовать уже для других операций.

Таким образом, использование конструкций async и await на языке C# требуется для построения асинхронного метода, выполняющего некие операции типа Task<T> с результатом типа T или вовсе без него.

После изменения асинхронного метода требуется построить для него машину состояний:

Листинг 3. Машина состояний для асинхронного метода Main на языке C#

```
[CompilerGenerated]
private sealed class <Main>d__0 : IAsyncStateMachine
{
    public int <>1__state;

    public AsyncTaskMethodBuilder <>t__builder;

    private TaskAwaiter <>u__1;

    private void MoveNext()
    {
        …
    }
    void IAsyncStateMachine.MoveNext()
    {
        this.MoveNext();
    }
```

```
     private void SetStateMachine([Nullable(1)] IAsyncStateMa-
chine stateMachine)
     {
    …
     }

…
}
```

Из данного кода следует, что компилятор сгенерировал для асинхронного метода Main структуру <Main>d__0, а также на стеке инициализировал экземпляр данной структуры. Эта структура является машиной состояния для асинхронного метода. Она содержит всю преобразованную логику, которую написал разработчик, в том числе поля, позволяющие отследить текущую позицию, которая соответствует состоянию асинхронного метода. Кроме того, в каждом состоянии содержатся текущие значения всех переменных, которые хранятся в полях класса машины состояний. Это нужно для того, чтобы при переходе из состояния в состояние с помощью метода MoveNext сохранялись и правильно обрабатывались значения данных переменных.

Стоит более подробно обсудить метод MoveNext:

Листинг 4. Внутренняя реализация метода MoveNext() на языке C#

```
private void MoveNext()
          {
               int num = <>1__state;
               try
               {
                    TaskAwaiter awaiter;
                    if (num != 0)
                    {
                         Console.WriteLine("MainStart");
                         awaiter = Task.Delay(2000).GetAwaiter();
                         if (!awaiter.IsCompleted)
                         {
                              num = (<>1__state = 0);
                              <>u__1 = awaiter;
                              <Main>d__0 <Main>d__ = this;
                              <>t__builder.AwaitUnsafeOnCom-
pleted(ref awaiter, ref <Main>d__);
                              return;
                         }
                    }
                    else
```

```
            {
                awaiter = <>u__1;
                <>u__1 = default(TaskAwaiter);
                num = (<>1__state = -1);
            }
            awaiter.GetResult();
            Console.WriteLine("MainEnd");
        }
        catch (Exception exception)
        {
            <>1__state = -2;
            <>t__builder.SetException(exception);
            return;
        }
        <>1__state = -2;
        <>t__builder.SetResult();
    }
```

Данный метод является ключевым. Как и говорилось выше, метод MoveNext() отвечает за работу с состояниями машины состояний, то есть содержит основную логику работы нашего асинхронного метода Main.

В блоке try метода MoveNext() состояния представлены в виде условных конструкций. В блоке catch отлавливаются исключения. При переходе в финальное состояние с помощью переменной типа TaskAwaiter<T> возвращается результат с помощью метода GetResult() при условии, что до этого никаких ошибок не возникло. При возникновении исключений происходит завершение метода MoveNext() через билдер с помощью метода SetException.

Не зависимо от того, как завершилось выполнение (с ошибкой или без), с помощью билдера вызывается метод SetResult.

Стоит отметить, что любой Task может находиться в одном из трех возможных конечных состояний:

- Успех
- Неудача
- Отмена

С успехом и неудачей (завершение с ошибкой) все понятно. Если задачу отменить, то в этом случае создается CancellationToken, чтобы выйти из задачи

без исключения. При этом гарантируется, что метод не завершит свою работу до тех пор, пока отмена не будет обработана соответствующим образом.

Таким образом, с помощью всех описанных выше преобразований мы получаем асинхронный код.

3. Дополнение грамматики языка PascalABC.NET

После изучения предметной области нужно было добавить ключевые слова async и await в грамматику языка PascalABC.NET. Кратко опишу изменения, внесенные в грамматику.

Конструкция async теперь может указываться перед процедурами и функциями:

Листинг 5. Async перед процедурами и функциями

```
| tkAsync method_procfunc_header
{
     ($2 as procedure_header).IsAsync = true;
     $$ = $2;
}
```

А также перед методами классов, в том числе и перед статическими методами:

Листинг 6. Async перед методами класса

```
| tkAsync class_or_static method_procfunc_header
        {
              ($3 as procedure_header).class_keyword = true;
              ($3 as procedure_header).IsAsync = true;
              $$ = $3;
        }
…
| tkAsync proc_func_decl_noclass
        {
              ($2 as procedure_definition).proc_header.IsAsync =
true;
              $$ = $2;
        }
    | tkAsync class_or_static proc_func_decl_noclass
        {
          ($3 as procedure_definition).proc_header.IsAsync =
true;
              ($3 as procedure_defini-
tion).proc_header.class_keyword = true;
              $$ = $3;
        }
    | class_or_static tkAsync proc_func_decl_noclass
        {
          ($3 as procedure_definition).proc_header.IsAsync =
true;
```

```
                ($3 as procedure_defini-
tion).proc_header.class_keyword = true;
                $$ = $3;
            }
```

Для конструкции await потребовалось с помощью программы NodesGenerator добавить новые узлы в синтаксическое дерево:

Рис.1 Новый узел await_node

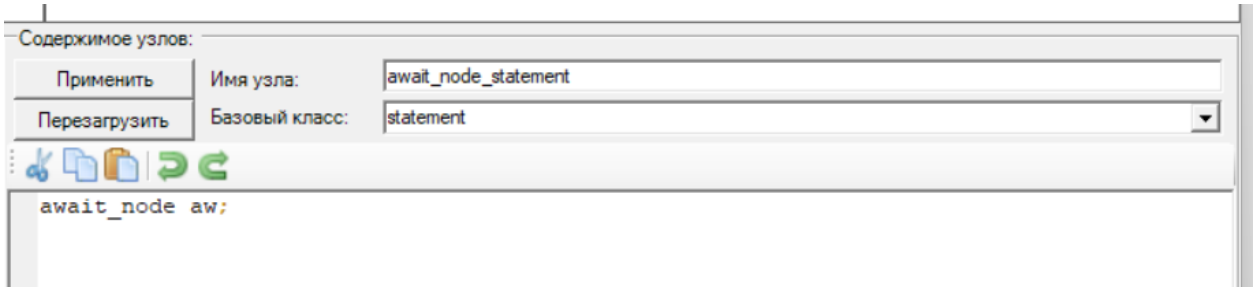

Рис.2 Новый узел await_node_statement

С соответствующими изменениями грамматики:

Листинг 7. Await может быть как expression, так и statement

```
expr_dq
    : relop_expr
        { $$ = $1; }
    | tkAwait relop_expr
        { $$ = new await_node($2 as expression, @$); }

    ...

  | proc_call
        { $$ = $1; }
    | tkAwait expr_l1
        { $$ = new await_node_statement(new
await_node($2,@$),@$); }
```

Теперь появилась возможность создавать асинхронные методы с конструкциями async и await:

Листинг 8. Пример асинхронного метода на языке PascalABC.NET

```
type
  MyClass = class
  private
  public
      static p1:= 101;
      as1:= 2;
    async function Fun2() : Task<integer>;
  end;
async function MyClass.Fun2() : Task<integer>;
begin
   var tt := Task.Run(() -> 5);
   println('StartFun2');
   var a:= await tt;
    println('EndFun2');
  Result:= 42;

end;
```

Осталось только обработать логику работы данных методов с помощью специально создаваемой под каждый асинхронный метод машины состояний.

4. Построение машины состояний для языка PascalABC.NET

Для успешной навигации через различные этапы асинхронного выполнения компилятор генерирует конечный автомат (машину состояний), который переносит наш асинхронный код в класс машины состояний. Любая задача, запускаемая в конечном автомате, может находиться в одном из четырех различных состояний:

- Процесс выполнения еще не начался
- Выполнение
- Приостановлено
- Завершено (это состояние включает как успешное, так и ошибочное завершение)

Таким образом можно управлять поведением выполнения асинхронных операций, что позволяет эффективно координировать и обрабатывать сложные асинхронные задачи, обеспечивая более надежное и предсказуемое выполнение программы.

Состояния «выполнение» так же, как и «не запущено» обозначаются как -1, а состояние «завершено» обозначается как -2. В противном случае можно считать состояние «приостановленным».

Машину состояний любого асинхронного метода можно описать как следующий конечный автомат:

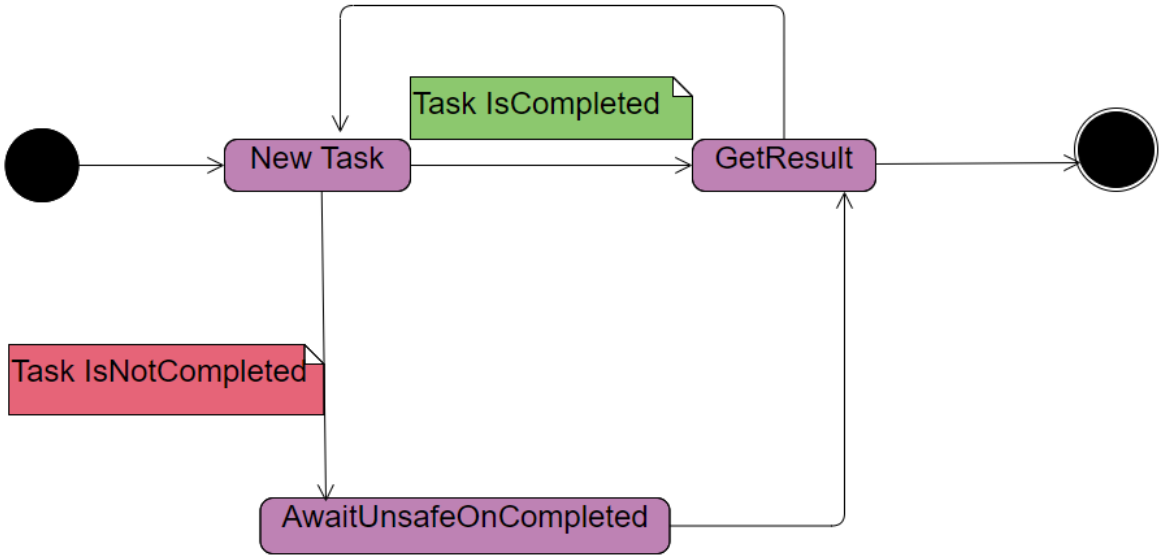

Рис.3 Машина состояний

Для того, чтобы построить машину состояний на языке PascalABC.NET аналогично тому, как она была реализована на языке C# требуется создать класс StateMachine и реализовать методы MoveNext() и SetStateMachine() интерфейса ***IAsyncStateMachine.*** Мною был написан код, генерирующий данную StateMachine. Минимальная версия машины состояний на языке PascalABC.NET выглядит следующим образом:

Листинг 9. Класс машины состояний на языке PascalABC.NET

```
uses System.Runtime.CompilerServices;

type
  StateMachine = class(IAsyncStateMachine)
  private
  public
    state: integer;
    tbuilder: AsyncVoidMethodBuilder;
    procedure MoveNext();
    procedure SetStateMachine(stateMachine: System.Runtime.Com-
pilerServices.IAsyncStateMachine);
  end;
```

- В переменной state содержится номер текущего состояний машины состояний.
- tbuilder типа AsyncVoidMethodBuilder представляет собой структуру, способную порождать объект данного типа, который содержит свойство задачи (Task). Эта структура обладает способностью завершать задачу либо успешно с возвращаемым значением, если это требуется (путем вызова SetResult), либо с исключением (путем вызова SetException). Кроме того, билдер умеет обрабатывать присоединение продолжений к задаче, ожидающей завершения другой операции с помощью методов AwaitOnCompleted или AwaitUnsafeOnCompleted.
- При запуске машины состояний вызывается метод MoveNext(), а также при переходе из одного состояния в другое в зависимости от количества и расстановки await.
- Метод SetStateMachine() нужен для создания и запуска машины состояний:

Листинг 10. Метод SetStateMachine на языке PascalABC.NET

```
procedure StateMachine.SetStateMachine(stateMachine: Sys-
tem.Runtime.CompilerServices.IAsyncStateMachine);
begin
  tbuilder := AsyncVoidMethodBuilder.Create();
  tbuilder.SetStateMachine(stateMachine);
end;
```

Рассмотрим, более подробно, как в моей реализации происходит построение машины состояний для некоторой асинхронной функции:

Листинг 11. Простейшая асинхронная функция на языке PascalABC.NET

```
async function MainMethod() : Task<integer>;
  begin
     var t := Task.Run(() -> 5);
     await t;
   end;
```

Функция MainMethod будет преобразована подобно тому, как это бы происходило на языке C#:

Листинг 12. Преобразование асинхронной функции на языке PascalABC.NET

```
async function MainMethod() : Task<integer>;
begin
  var st:= new StateMachine(); // инициализация машины состояний
  st.state := 1; // начальное состояние
 st.tbuilder := AsyncTaskMethodBuilder&<integer>.Create();
 st.tbuilder.Start(st);
 Result:= st.tbuilder.Task;
end;
```

После инициализации машины состояний и присваивания переменной state начального состояния равного 1 требуется вызвать метод AsyncTaskMethodBuilder.Create(), который создает экземпляр асинхронного метода. А метод Start() нужен для запуска метода MoveNext():

Листинг 13. Генерация метода MoveNext() на языке PascalABC.NET

```
procedure StateMachine.MoveNext();
begin
  try
   // различные состояния
  except
    state := -2;
    tbuilder.SetException(new Exception('MoveNextException'));
    exit;
  end;
  state := -2;
  tbuilder.SetResult();
end;
```

Если исключение, брошенное в блоке try, не будет обработано, тогда задача завершится с этим исключением. В случае успешного завершения асинхронного метода (эквивалентного синхронному методу, завершенному без ошибок), задача также завершится успешно. В обоих этих сценариях происходит установка состояния конечного автомата, указывающего на его завершение.

Листинг 14. Блок try метода MoveNext() на языке PascalABC.NET

```
try
…
    if (state = 1) then
```

```
begin
t:= Task.Run(() -> 5);
  awaiter := t.GetAwaiter;
  if (awaiter.IsCompleted = False) then
  begin
    state:=2;
    u1:= awaiter;
    var temp := self;
    tbuilder.AwaitUnsafeOnCompleted(awaiter, temp);
      exit;
  end;
  goto lab1;
end;
if (state = 2) then
begin
      state:= -1;
       goto lab1;
end;
if (state = 3) then
begin
      state:= -1;
       goto lab2;
end;
lab1:
awaiter.GetResult();
…
lab2:
awaiter.GetResult();
```

Опишем более подробно работу метода MoveNext:

- Первоначальная проверка состояния (state = 1 это начальное состояние)
- Дальнейшее выполнение конечного автомата зависит от результата выполнения текущего Task. Проверка происходит с помощью метода IsCompleted.
- Происходит установка состояния, указывающего, что задача запущена и/или выполняется.
- Если задача уже завершена, то это равносильно синхронному коду, а иначе
- Запускается метод AwaitUnsafeOnCompleted() для дальнейшего возобновления с прежней точки прерывания.

- После завершения указанного ожидания конечный автомат планирует перейти к следующему состоянию.
- Переход к финальному состоянию. Метод завершается, и Task возвращается со статусом завершения. Если тип возвращаемого значения равен Task <TResult>, то также возвращаем результат и запоминаем его с помощью метода awaiter.GetResult().

Интересная часть заключается в том, что сам метод MoveNext() ничего не возвращает, поскольку изменяет состояние по ссылке. По своей конструкции он полагается на завершение ожидания(awaiter).

Таким образом, метод MoveNext() отвечает за управление состоянием: проверяет, завершено ли ожидание или нет, и изменяет состояние конечного автомата и/или задачи.

Конечный результат моей реализации для построения базового класса StateMachine для языка PascalABC.NET:

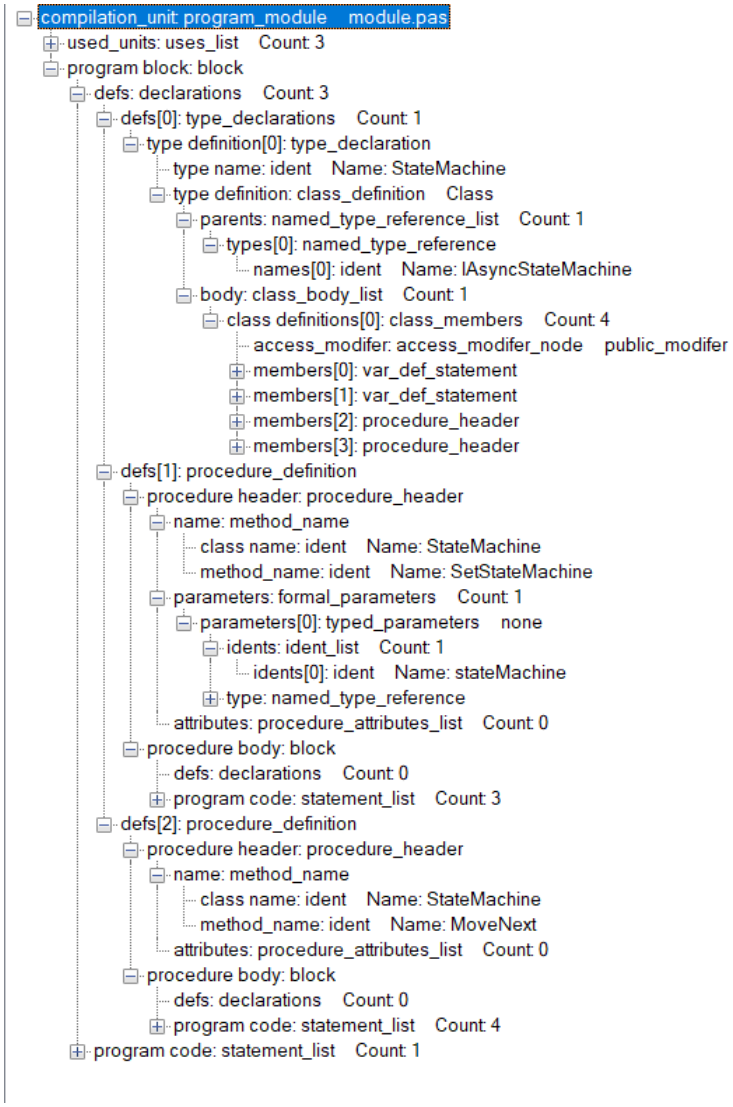

Рис. 4 Сгенерированная машина состояний в дереве программы на языке Pascal.ABC.NET

5. Захват локальных переменных

При работе с асинхронными методами зачастую возникает проблема с переменными. Суть данной проблемы заключается в том, что в методе MoveNext() при переходе от одного состояния в другое вызывается метод AwaitUnsafeOnCompleted(см. Рис.3). Вызывается данный метод, потому что код находящийся перед await еще не успел выполниться, поэтому нам нужно сохранить текущее состояние машины (со значениями всех полей класса State-Machine), чтобы потом при переходе в новое состояние значения переменных класса не обнулились. Если бы код успел выполниться до вызова метода Is-Completed(), то программа выполнилась бы аналогично синхронной.

Существуют различные типы переменных, которые нужно сохранить для дальнейшего восстановления при переходе в новое состояние машины:

- Локальные переменные асинхронного метода
- Параметры асинхронного метода
- Переменные внутри конструкций (if, for, while, case и т. д.)
- Поля класса, внутри которого вызывается асинхронный метод
- Статические поля класса, внутри которого вызывается асинхронный метод
- Глобальные переменные и прочие нетривиальные случаи

Начнем с описания локальных переменных.

Локальные переменные бывают двух типов:

1) Только с объявлением
2) С объявлением и инициализацией

Для того чтобы сохранить значения локальных переменных первого типа, необходимо объявить их переменными класса StateMachine.

Если же переменная является переменной второго типа (уже инициализирована), тогда для объявления ее переменной класса необходимо сначала узнать ее тип.

Проще всего это сделать по первому присваиванию. Для этого достаточно возвести флаг first_assignment_defines_type в true:

Листинг 15. Пример с first_assignment_defines_type на языке PascalABC.NET

```
var ass = new assign(item, pp.var_def.inital_value,
Operators.Assignment,item.source_context);
ass.first_assignment_defines_type = true;
```

После этого все локальные переменные необходимо переименовать, чтобы не возникло конфликтов имен. К именам переменных я решил добавлять приписку ***@awvar@_n,*** где n – это порядковый номер.

Таким образом я получаю следующий сгенерированный код с захватом локальных переменных:

Листинг 16. Захват локальных переменных на языке PascalABC.NET

```
type
  StateMachine = class(IAsyncStateMachine)
  private
  u1 : TaskAwaiter;
  u2 : TaskAwaiter;
  public
    state: integer; // состояние
    tbuilder: AsyncTaskMethodBuilder<integer>;
    ttt : Task<integer>;
    @awvar@_1_b : integer;
    @awvar@_2_c :  string; // тип изменится после первого при-
сваивания
    procedure MoveNext();
    procedure SetStateMachine(stateMachine: System.Runtime.Com-
pilerServices.IAsyncStateMachine);
  end;
…
procedure StateMachine.MoveNext();
label lab1,lab2;
begin
  try
    var t := system.Threading.Tasks.Task.Delay(3000);
    var awvar@_1_b:= 2;
    @awvar@_2_c := 5; // первое присваивание(значит нужно заме-
нить тип string на тип integer в объявлении)
    if (state = 1) then
    begin
    …
```

5.1. Захват параметров асинхронного метода

Для того чтобы захватить параметр асинхронного метода необходимо сначала объявить этот параметр полем класса StateMachine(точно также, как локальные переменные первого типа). После этого нужно в самом асинхронном методе скопировать значение этого параметра:

Листинг 17. Захват локальных параметров асинхронного метода на языке PascalABC.NET

```
public static void AsyncProcedure(int par)
    {
        @StateMachine@_2 @StateMachine@_ = new @StateMa-
chine@_2();
        @StateMachine@_.state = 1;
        @StateMachine@_.par= par;//сохранение значения параметра
        @StateMachine@_.tbuilder = AsyncVoidMethodBuilder.Cre-
ate();
        @StateMachine@_.tbuilder.Start(ref @StateMachine@_);
    }
```

А затем необходимо переименовать все параметры так, как это было сделано для локальных переменных.

5.2. Lowering

Для того чтобы захватить переменные внутри конструкций, необходимо в начале линеаризовать код асинхронного метода. Делается это с помощью алгоритмов Lowering.

Все алгоритмы сходятся в том, что нужно заменить конструкции (for, while, case …) на линейный код без блоков с помощью операторов if и goto.

Данная техника уже была реализована и использована в классе LoweringVisitor проекта PascalABC.NET для оператора yield, поэтому я использовал уже готовые алгоритмы, и в написанном мною классе LoweringAsyncVisitor адаптировал каждый из них под оператор await.

Таким образом, для каждой программы, содержащей оператор await, сначала будет выполняться Lowering для получения линейного кода, а затем переменные, находящиеся внутри блочных конструкций, будут захватываться также, как и локальные.

5.3. Захват полей класса, внутри которого вызывается асинхронный метод

Для реализации захвата полей класса необходимо вначале в асинхронном методе скопировать ссылку на этот класс и поместить ее в поле класса машины состояний:

Листинг 18. Преобразование асинхронного метода P3 класса MyClass на языке PascalABC.NET

```
     public class MyClass
{
…
public void P3(int par)
     {
          @StateMachine@_2 @StateMachine@_ = new @StateMa-
chine@_2();
          @StateMachine@_.state = 1;
          @StateMachine@_.par = par;
          @StateMachine@_.tbuilder = AsyncVoidMethodBuilder.Cre-
ate();
          @StateMachine@_.@awclass = this; //копируем ссылку на
MyClass
          @StateMachine@_.tbuilder.Start(ref @StateMachine@_);
     }
```

После этого нужно все переменные этого класса вызывать через @awclass - поле StateMachine, содержащее объект MyClass .

Листинг 19. Захват полей класса, внутри которого вызывается асинхронный метод на языке PascalABC.NET

```
public class @StateMachine@_1 : IAsyncStateMachine
{
     internal TaskAwaiter<int> @aw@_1;

     internal MyClass @awclass;

     public int state;

     public AsyncTaskMethodBuilder<int> tbuilder;

     public int @aw_res;

     public Task<int> tt;

     public void MoveNext()
     {
          try
```

```
{
     if (state == 1)
     {
          PABCSystem.PABCSystem.Println(@awclass.as1);
```

5.4. Захват статических полей класса, внутри которого вызывается асинхронный метод

Для статических полей достаточно переименовать все переменные типа <название переменной> в переменные типа MyClass.<название переменной>

Листинг 20. Захват статических полей класса, внутри которого вызывается асинхронный метод на языке PascalABC.NET

```
public class MyClass
{
    public static int p1;
…
}
public class @StateMachine@_2 : IAsyncStateMachine
{
…
public void MoveNext()
    {
MyClass.p1 += 5;
…
}
```

5.5. Захват глобальных переменных

В эту категорию попадают все имена, не относящиеся к другим 3 категориям (локальные переменные, формальные параметры и имена класса). Специального алгоритма захвата таких имен не предусмотрено.

6. Пример работы

Рассмотрим пример работы асинхронных методов из статьи Microsoft [1].

Приготовление завтрака в асинхронном режиме является хорошим примером того, как один человек (или поток) может эффективно управлять множеством задач без необходимости параллельного выполнения. Повар может асинхронно готовить завтрак, начиная следующую задачу до завершения текущей. Процесс приготовления не зависит от наличия посторонних наблюдателей и продолжается плавно: когда сковорода разогреется, можно перейти к жарке бекона, а потом — к поджарке хлеба с помощью тостера. Это позволяет эффективно использовать ресурсы и время одного человека (или потока).

В отличие от асинхронности, параллельный алгоритм требует отдельных поваров (или потоков) для каждой задачи, например, один готовит яйца, другой — бекон, и так далее. Каждый из них фокусируется исключительно на своей задаче, и выполнение каждого шага блокируется синхронно, чтобы дождаться готовности других компонентов блюда.

Листинг 21. Пример синхронного выполнения задач при приготовлении бекона

```
type
  Bacon = class
  public
    static function FryBacon(slices: integer): Bacon;
  end;

static function Bacon.FryBacon(slices: integer): Bacon;
begin
  Println($'putting {slices} slices of bacon in the pan');
  Println('cooking first side of bacon...');
  Task.Delay(3000).Wait();
  for var slice := 0 to slices do
  begin
    Println('flipping a slice of bacon');
  end;
  Println('cooking the second side of bacon...');
  Task.Delay(3000).Wait();
  Println('Put bacon on plate');
```

```
  Result := new Bacon();
end;
```

Мы создаем задачи, и вынуждены дожидаться их завершения. Пока не закончена текущая задача, к следующей переходить нельзя:

Листинг 22. Пример синхронного выполнения задач при приготовлении завтрака

```
begin
  var stopwatch := new Stopwatch();
  stopwatch.Start();
   var cup := Coffee.PourCoffee();
 Println('coffee is ready');

 var eggs := Egg.FryEggs(2);
 Println('eggs are ready');

 var bacon_ := Bacon.FryBacon(3);
 Println('bacon is ready');

 var toast_ := Toast.ToastBread(2);
 ApplyButter(toast_);
 ApplyJam(toast_);
 Println('toast is ready');
  var oj := Juice.PourOJ();
  Println('oj is ready');
  Println('Breakfast is ready!');
  stopwatch.Stop();
  println($'Total time taken: {stopwatch.Elapsed}');
end.
```

Синхронное приготовление завтрака заняло примерно 30 минут, поскольку общее время выполнения каждой задачи равняется сумме.

Рис. 5 Синхронное приготовление завтрака

Компьютеры интерпретируют инструкции по-разному, блокируя выполнение каждого задания до его завершения перед переходом к следующему. Этот подход приводит к неудовлетворительному результату при приготовлении завтрака, так как последующие задания ждут завершения предыдущих, что замедляет процесс.

Это приводит к увеличению времени приготовления завтрака.

Листинг 23. Плохой пример использования await

```
begin
  var stopwatch := new Stopwatch();
  stopwatch.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggs := Egg.FryEggs(2);
  await eggs;
  Println('eggs are ready');
  var bacontask := Bacon.FryBacon(3);
  await bacontask;
```

```
  Println('bacon is ready');
  var toast_ := Toast.ToastBread(2);
  await toast_;
  ApplyButter(toast_.Result);
  ApplyJam(toast_.Result);
  Println('toast is ready');
  var oj := Juice.PourOJ();
  Println('oj is ready');
  Println('Breakfast is ready!');
  stopwatch.Stop();
  println($'Total time taken: {stopwatch.Elapsed}');
end.
```

Предыдущий пример кода иллюстрирует неправильную практику: мы ожидаем завершение задачи сразу же после ее запуска, что аналогично выполнению синхронного кода. В таком коде текущий поток блокируется от выполнения другой работы и не может быть прерван во время выполнения задачи. Это подобно тому, как вы наблюдаете за тостером, положив в него хлеб и не отвлекаясь на какие-то другие задачи, пока тостер не приготовит тосты.

В то же время, если запустить задачи, не дожидаясь их завершения, то время приготовления завтрака уменьшится:

Листинг 24. Хороший пример использования await

```
begin
  var stopwatch := new Stopwatch();
  stopwatch.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggsTask := Egg.FryEggs(2);
  var baconTask := Bacon.FryBacon(3);
  var toastTask := Toast.ToastBread(2);

var toast:= await toastTask;
  ApplyButter(toast);
  ApplyJam(toast);
  Println('toast is ready');
   var oj := Juice.PourOJ();
  Println('oj is ready');

 var eggs := await eggsTask;
  Println('eggs are ready');

 var bacon := await baconTask;
  Println('bacon is ready');

  Println('Breakfast is ready!');
  stopwatch.Stop();
```

```
  println($'Total time taken: {stopwatch.Elapsed}');
end.
```

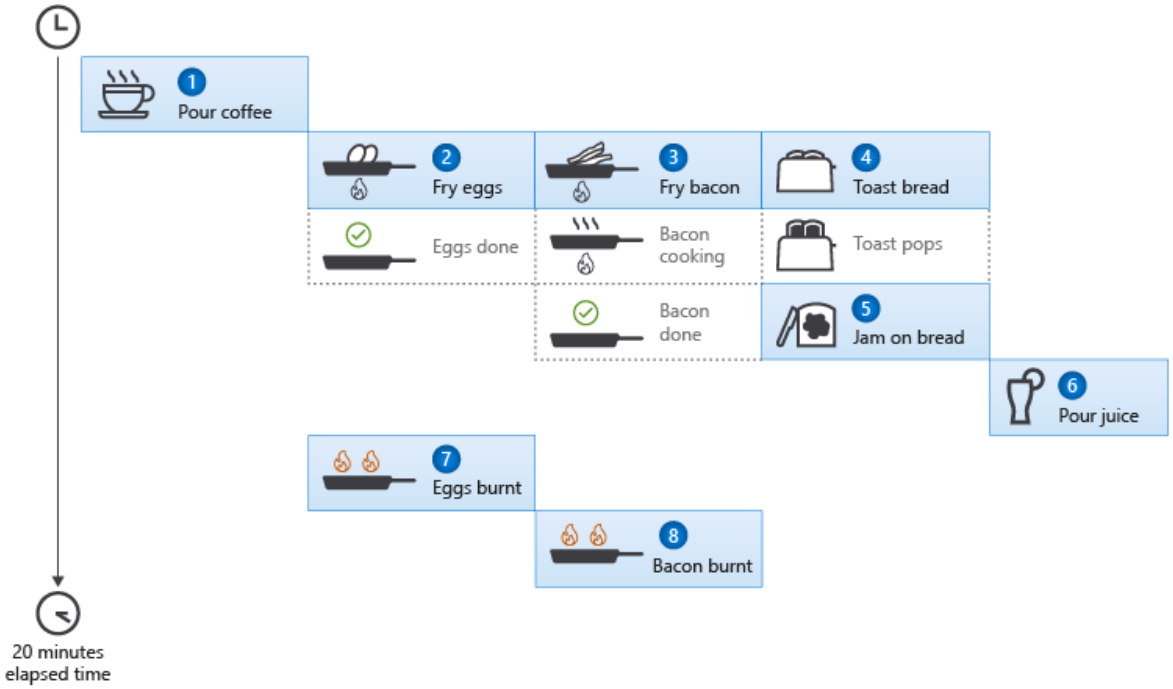

Рис. 6 Асинхронное приготовление завтрака

Асинхронная подготовка завтрака заняла около 20 минут, что позволило сэкономить время, так как различные задачи выполнялись параллельно.

Предыдущий код демонстрирует более эффективную работу. Вы запускаете все асинхронные задачи одновременно, ожидая результатов только по мере необходимости.

Если же вам не хочется заботиться о том, в какой последовательности запускать задачи или дожидаться их завершения, то можно воспользоваться методом Task.WhenAny:

Листинг 25. Асинхронное приготовление завтрака

```
begin
  var stopwatch := new Stopwatch();
  stopwatch.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggsTask := Egg.FryEggs(2);
  var baconTask := Bacon.FryBacon(3);
 var toastTask := MakeToastWithButterAndJamAsync(1);
  var breakfastTasks := new List<Task>();
```

```
  breakfastTasks.AddRange(Arr&<Task>(eggsTask,bacon-
Task,toastTask));

  while(breakfastTasks.Count > 0) do
  begin
     var finishedTask :=  await Task.WhenAny(breakfastTasks);
 if (finishedTask = eggsTask) then
 begin
     Println('eggs are ready');
 end;
if (finishedTask = baconTask) then
 begin
     Println('bacon is ready');
 end;
if (finishedTask = toastTask) then
 begin
     Println('toast is ready');
 end;
 await finishedTask;
 breakfastTasks.Remove(finishedTask);
  end;

  Println('Breakfast is ready!');
  stopwatch.Stop();
  println($'Total time taken: {stopwatch.Elapsed}');
end.
```

Этот окончательный код является асинхронным. Он более точно отражает то, как человек готовил бы завтрак.

Заключение

В результате работы были реализованы конструкции async и await для языка программирования PascalABC.NET.

Разработан алгоритм построения машины состояний для асинхронных методов с последующей обработкой их логики.

Реализован алгоритм захвата локальных переменных

Работа над проектом велась в открытом репозитории на GitHub[4] в основном в решении SyntaxTreeVisitors

Таким образом, на языке программирования PascalABC.NET появилась возможность написания асинхронных методов, работы с ними, используя различные конструкции, аналогично тому, как это делается на языке C#.

Литература

1. Пример работы Async/Await C#. – https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/ (дата обращения 14.10.2023).
2. Принцип работы Async/Await C#. – https://devblogs.microsoft.com/dotnet/how-async-await-really-works/ (дата обращения 08.11.2023).
3. Asynchronous Programming in C# 5.0 | Alex Davis
4. PascalABC.NET at GitHub [Электронный ресурс]. – https://github.com/pascalabcnet/pascalabcnet
5. Паттерны объектно-ориентированного проектирования | Гамма Эрих, Хелм Ричард
6. Документация по Task. – https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-8.0 (дата обращения 18.05.2023).

Приложение 1. Примеры кода

Рассмотрим реализацию примеров из статьи Microsoft [1]

на языке Pascal.ABC.NET:

Листинг 26. Синхронное приготовление завтрака на языке PascalABC.NET

```
uses System.Diagnostics;
uses System.Threading.Tasks;

type
  Bacon = class
  public
    static function FryBacon(slices: integer): Bacon;
  end;

static function Bacon.FryBacon(slices: integer): Bacon;
begin
  Println($'putting {slices} slices of bacon in the pan');
  Println('cooking first side of bacon...');
  Task.Delay(3000).Wait();
  for var slice := 0 to slices do
  begin
    Println('flipping a slice of bacon');
  end;
  Println('cooking the second side of bacon...');
  Task.Delay(3000).Wait();
  Println('Put bacon on plate');

  Result := new Bacon();
end;

type
  Coffee = class
  public
    static function PourCoffee(): Coffee;
  end;

static function Coffee.PourCoffee(): Coffee;
begin
  Println('Pouring coffee');
  Result := new Coffee();
end;

type
  Egg = class
  public
    static function FryEggs(howMany: integer): Egg;
  end;

static function Egg.FryEggs(howMany: integer): Egg;
begin
```

```
  Println('Warming the egg pan...');
  Task.Delay(3000).Wait();
  Println($'cracking {howMany} eggs');
  Println('cooking the eggs ...');
  Task.Delay(3000).Wait();
  Println('Put eggs on plate');
  Result := new Egg();
end;

type
  Juice = class
  public
    static function PourOj(): Juice;
  end;

static function Juice.PourOj(): Juice;
begin
  Println('Pouring orange juice');
  Result := new Juice();
end;

type
  Toast = class
  public
    static function ToastBread(slices: integer): Toast;
  end;

static function Toast.ToastBread(slices: integer): Toast;
begin
  for var slice := 0 to slices do
  begin
    Println('Putting a slice of bread in the toaster');
  end;
  Println('Start toasting...');
  Task.Delay(3000).Wait();
  Println('Remove toast from toaster');

  Result := new Toast();
end;

procedure ApplyJam(toast: Toast);
begin
  Println('Putting jam on the toast');
end;

procedure ApplyButter(toast: Toast);
begin
  Println('Putting butter on the toast');
end;

begin
  var stopwatch := new Stopwatch();
  stopwatch.Start();
```

```
   var cup := Coffee.PourCoffee();
 Println('coffee is ready');

 var eggs := Egg.FryEggs(2);
 Println('eggs are ready');

 var bacon_ := Bacon.FryBacon(3);
 Println('bacon is ready');

 var toast_ := Toast.ToastBread(2);
 ApplyButter(toast_);
 ApplyJam(toast_);
 Println('toast is ready');
  var oj := Juice.PourOJ();
  Println('oj is ready');
  Println('Breakfast is ready!');
  stopwatch.Stop();
  println($'Total time taken: {stopwatch.Elapsed}');
end.
```

Листинг 27. Неправильное использование await на языке PascalABC.NET

```
uses System.Diagnostics;
uses System.Threading.Tasks;

type
  Bacon = class
  public
    async static function FryBaconAsync(slices: integer):
Task<Bacon>;
  end;

async static function Bacon.FryBaconAsync(slices: integer):
Task<Bacon>;
begin
  Println($'putting {slices} slices of bacon in the pan');
  Println('cooking first side of bacon...');
  await Task.Delay(3000);
  for var slice := 1 to slices do
  begin
    Println('flipping a slice of bacon');
  end;
  Println('cooking the second side of bacon...');
 await Task.Delay(3000);
  Println('Put bacon on plate');

  Result := new Bacon();
end;

type
  Coffee = class
  public
```

```
    static function PourCoffee(): Coffee;
  end;

static function Coffee.PourCoffee(): Coffee;
begin
  Println('Pouring coffee');
  Result := new Coffee();
end;

type
  Egg = class
  public
    async static function FryEggsAsync(howMany: integer) :
Task<Egg>;
  end;

async static function Egg.FryEggsAsync(howMany: integer):
Task<Egg>;
begin
  Println('Warming the egg pan...');
   await Task.Delay(3000);
  Println($'cracking {howMany} eggs');
  Println('cooking the eggs ...');
   await Task.Delay(3000);
  Println('Put eggs on plate');
  Result := new Egg();
end;

type
  Juice = class
  public
    static function PourOj(): Juice;
  end;

static function Juice.PourOj(): Juice;
begin
  Println('Pouring orange juice');
  Result := new Juice();
end;

type
  Toast = class
  public
    async static function ToastBreadAsync(slices_: integer):
Task<Toast>;
  end;

async static function Toast.ToastBreadAsync(slices_: integer):
Task<Toast>;
begin
  for var slice := 1 to slices_ do
  begin
    Println('Putting a slice of bread in the toaster');
```

```
  end;
  Println('Start toasting...');
  await Task.Delay(3000);
  Println('Remove toast from toaster');

  Result := new Toast();
end;

procedure ApplyJam(toast: Toast);
begin
  Println('Putting jam on the toast');
end;

procedure ApplyButter(toast: Toast);
begin
  Println('Putting butter on the toast');
end;

begin
  var sw := new Stopwatch();
  sw.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggsTask := await Egg.FryEggsAsync(2);
  Println('eggs are ready');
  var baconTask := await Bacon.FryBaconAsync(3);
  Println('bacon is ready');
  var toastTask := await Toast.ToastBreadAsync(1);
  ApplyButter(toastTask);
  ApplyJam(toastTask);
  Println('toast is ready');
  var oj := Juice.PourOJ();
  Println('oj is ready');
  Println('Breakfast is ready!');
  sw.Stop();
  println($'Total time taken: {sw.Elapsed}');
end.
```

Листинг 28. Правильное использование await на языке PascalABC.NET

```
uses System.Diagnostics;
uses System.Threading.Tasks;

type
  Bacon = class
  public
    async static function FryBaconAsync(slices: integer):
Task<Bacon>;
  end;

async static function Bacon.FryBaconAsync(slices: integer):
Task<Bacon>;
```

```
begin
  Println($'putting {slices} slices of bacon in the pan');
  Println('cooking first side of bacon...');
  await Task.Delay(3000);
  for var slice := 1 to slices do
  begin
    Println('flipping a slice of bacon');
  end;
  Println('cooking the second side of bacon...');
 await Task.Delay(3000);
  Println('Put bacon on plate');

  Result := new Bacon();
end;

type
  Coffee = class
  public
    static function PourCoffee(): Coffee;
  end;

static function Coffee.PourCoffee(): Coffee;
begin
  Println('Pouring coffee');
  Result := new Coffee();
end;

type
  Egg = class
  public
    async static function FryEggsAsync(howMany: integer) :
Task<Egg>;
  end;

async static function Egg.FryEggsAsync(howMany: integer):
Task<Egg>;
begin
  Println('Warming the egg pan...');
   await Task.Delay(3000);
  Println($'cracking {howMany} eggs');
  Println('cooking the eggs ...');
   await Task.Delay(3000);
  Println('Put eggs on plate');
  Result := new Egg();
end;

type
  Juice = class
  public
    static function PourOj(): Juice;
  end;

static function Juice.PourOj(): Juice;
```

```
begin
  Println('Pouring orange juice');
  Result := new Juice();
end;

type
  Toast = class
  public
    async static function ToastBreadAsync(slices_: integer):
Task<Toast>;
  end;

async static function Toast.ToastBreadAsync(slices_: integer):
Task<Toast>;
begin
  for var slice := 1 to slices_ do
  begin
    Println('Putting a slice of bread in the toaster');
  end;
  Println('Start toasting...');
  await Task.Delay(3000);
  Println('Remove toast from toaster');

  Result := new Toast();
end;

procedure ApplyJam(toast: Toast);
begin
  Println('Putting jam on the toast');
end;

procedure ApplyButter(toast: Toast);
begin
  Println('Putting butter on the toast');
end;

begin
  var sw := new Stopwatch();
  sw.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggsTask := Egg.FryEggsAsync(2);
  var baconTask := Bacon.FryBaconAsync(3);
  var toastTask := Toast.ToastBreadAsync(1);

var toast_:= await toastTask;
  ApplyButter(toast_);
  ApplyJam(toast_);
  Println('toast is ready');
   var oj := Juice.PourOJ();
  Println('oj is ready');

 var eggs := await eggsTask;
```

```
  Println('eggs are ready');

var bacon_ := await baconTask;
  Println('bacon is ready');
  Println('Breakfast is ready!');
  sw.Stop();
  println($'Total time taken: {sw.Elapsed}');
end.
```

Листинг 29. Финальная версия асинхронного кода на языке PascalABC.NET

```
uses System.Diagnostics;
uses System.Threading.Tasks;

type
  Bacon = class
  public
    async static function FryBacon(slices: integer): Task<Ba-
con>;
  end;

async static function Bacon.FryBacon(slices: integer): Task<Ba-
con>;
begin
  Println($'putting {slices} slices of bacon in the pan');
  Println('cooking first side of bacon...');
  await Task.Delay(3000);
  for var slice := 1 to slices do
  begin
    Println('flipping a slice of bacon');
  end;
  Println('cooking the second side of bacon...');
 await Task.Delay(3000);
  Println('Put bacon on plate');

  Result := new Bacon();
end;

type
  Coffee = class
  public
    static function PourCoffee(): Coffee;
  end;

static function Coffee.PourCoffee(): Coffee;
begin
  Println('Pouring coffee');
  Result := new Coffee();
end;

type
  Egg = class
```

```
  public
    async static function FryEggs(howMany: integer) : Task<Egg>;
  end;

async static function Egg.FryEggs(howMany: integer): Task<Egg>;
begin
  Println('Warming the egg pan...');
   await Task.Delay(3000);
  Println($'cracking {howMany} eggs');
  Println('cooking the eggs ...');
   await Task.Delay(3000);
  Println('Put eggs on plate');
  Result := new Egg();
end;

type
  Juice = class
  public
    static function PourOj(): Juice;
  end;

static function Juice.PourOj(): Juice;
begin
  Println('Pouring orange juice');
  Result := new Juice();
end;

type
  Toast = class
  public
    async static function ToastBread(slices_: integer):
Task<Toast>;
  end;

async static function Toast.ToastBread(slices_: integer):
Task<Toast>;
begin
  for var slice := 1 to slices_ do
  begin
    Println('Putting a slice of bread in the toaster');
  end;
  Println('Start toasting...');
  await Task.Delay(3000);
  Println('Remove toast from toaster');

  Result := new Toast();
end;

procedure ApplyJam(toast: Toast);
begin
  Println('Putting jam on the toast');
end;
```

```
procedure ApplyButter(toast: Toast);
begin
  Println('Putting butter on the toast');
end;
async function  MakeToastWithButterAndJamAsync(number:inte-
ger):Task<Toast>;
begin
 var toastTask := await Toast.ToastBread(number);
   ApplyButter(toastTask);
   ApplyJam(toastTask);
   Result:= toastTask;
end;
begin
  var sw := new Stopwatch();
  sw.Start();
  var cup := Coffee.PourCoffee();
  Println('coffee is ready');
  var eggsTask := Egg.FryEggs(2);
  var baconTask := Bacon.FryBacon(3);
 var toastTask := MakeToastWithButterAndJamAsync(1);
  var breakfastTasks := new List<Task>();
  breakfastTasks.AddRange(Arr&<Task>(eggsTask,bacon-
Task,toastTask));

  while(breakfastTasks.Count > 0) do
  begin
     var finishedTask :=  await Task.WhenAny(breakfastTasks);
 if (finishedTask = eggsTask) then
 begin
     Println('eggs are ready');
 end;
if (finishedTask = baconTask) then
 begin
     Println('bacon is ready');
 end;
if (finishedTask = toastTask) then
 begin
     Println('toast is ready');
 end;
 await finishedTask;
 breakfastTasks.Remove(finishedTask);
  end;

  Println('Breakfast is ready!');
  sw.Stop();
  println($'Total time taken: {sw.Elapsed}');
end.
```