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

Филонов Алексей Константинович

Улучшенная оптимизация множественного присваивания в PascalABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
01.03.02— Прикладная математика и информатика

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

Допущено к защите:
заведующий кафедрой _____ Михалкович С.С.

Ростов-на-Дону – 2024

Оглавление

Постановка задачи

1. Разработать алгоритм оптимизации множественного присваивания для различных типов символов
2. Создать архитектуру внедрения указанного алгоритма в компилятор PascalABC.NET
3. Реализовать собственно алгоритм оптимизации множественного присваивания в компиляторе PascalABC.NET
4. Провести сравнительный анализ по количеству присваиваний и быстродействию для языков PascalABC.NET и C#

Введение

Множественное (кортежное) присваивание (Листинг 1) – сахарная конструкция, позволяющая присвоить значения сразу нескольким переменным.

Листинг 1. Пример множественного присваивания.

```
(a, b) := (b, a);
```

Кортежное присваивание, которое присутствует, например, в C#[1], Python[2], PascalABC.NET, позволяет программисту писать код короче и читаемее.

Однако, как и любая сахарная конструкция, множественное присваивание преобразуется компилятором в более простые конструкции, в данном случае – в подряд идущие отдельные присваивания. У компилятора PascalABC.NET количество генерируемых присваиваний равняется $2 * n$, где n – количество переменных в левом кортеже.

В работе [3] был разработан алгоритм для модельного языка программирования, оптимизирующий количество генерируемых присваиваний. Верхняя грань количества присваиваний – $2 * n$, нижняя грань – n.

Целью этой работы является внедрение алгоритма в компилятор PascalABC.NET и улучшение обработки алгоритмом различных типов символов (элементы массивов, выражения, составные имена, var-параметры, локальные и нелокальные переменные, указатели).

1. Типы символов

Тип символа определяет при каких условиях он может оказаться синонимом с другим символом, то есть быть одной ячейкой памяти. Например, в Листинге 2 `a[i]` и `b[j]` являются одной ячейкой памяти.

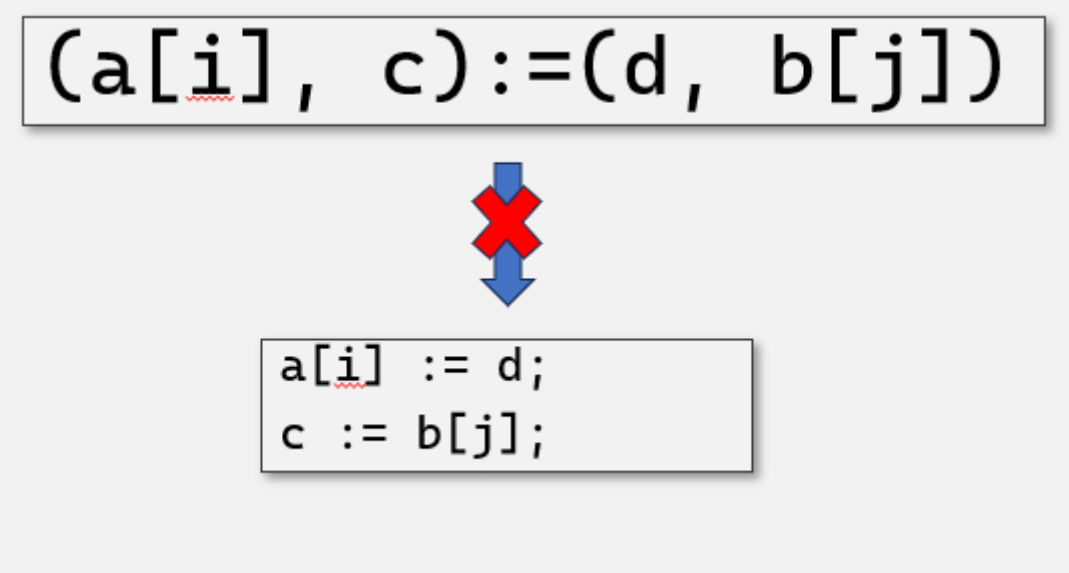
Листинг 2. Пример синонимов.

```
begin
  var a : array [1..3] of integer := (0, 1, 3);
  var b := a;
  var i := 1;
  var j := 1;
  println(a[i]);
  println(b[j]);
end.
```

При преобразовании кортежного присваивания в последовательные присваивания необходимо учитывать возможные синонимы, так как иначе возможны **некорректные присваивания** – присваивания, в результате которых переменной из левой части кортежного присваивания присваивается не соответствующее ей значение из правого кортежа. В дальнейшем под некорректными присваиваниями будет подразумеваться именно это определение.

На Рис. 1 приведен пример неправильного раскрытия множественного присваивания, так как если `a[i]` и `b[j]` являются одной ячейкой памяти, то в переменной `c` окажется значение переменной `d`, а не `b[j]`.

Рис. 1. Пример неправильного раскрытия кортежного присваивания



Кроме элементов массивов синонимами между собой могут оказаться составные имена, у которых совпадает последнее имя. Например, в Листинге 3 составные имена `aa.p, aa1.p, bb.f1.p` являются одной ячейкой памяти. Переменная из внешнего контекста тоже может оказаться синонимом с составным именем, если ее имя совпадает с последним именем составного имени (Листинг 4).

Листинг 3. Ситуация, когда составные имена оказываются синонимами

```
type
     A = class
          p : Integer;
     end;

     B = class
          f1 : A;

     end;

begin
     var aa := new A();
     var bb := new B();
     aa.p := 2;
     var aa1 := aa;
     bb.f1 := aa;
     println(aa.p);
     println(aa1.p);
     println(bb.f1.p);
end.
```

Листинг 4. Ситуация, когда составное имя оказывается синонимом с именем из внешнего контекста

```
type
     A = class
          p : Integer;
          procedure f(param: A);
          begin
               println(p);
               println(param.p);
          end;
     end;
begin
     var a1 := new A();
     a1.p := 3;
     a1.f(a1);
```

```
end.
```

Var-параметры могут быть одной ячейкой памяти с любым нелокальным именем.

Чтобы определить пришло ли простое имя из внешнего контекста или является ли оно var-параметром был написан визитор синтаксического дерева для поиска легковесной информации о символах – BindCollectLightSymInfo.

Визитор BindCollectLightSymInfo позволяет связывать использование символа с его определением. Для поиска используется функция bind, которой передается имя(ident). В процессе поиска строится древовидная структура пространств имен, которая заполняется и дополнятся при новых вызовах функции bind. В результате функция bind возвращает объект класса BindResult, который содержит в себе информацию о символе и путь ее поиска по дереву пространств имен. Информация включает в себя тип символа (переменная, параметр, имя функции, имя процедуры, имя класса, имя записи, имя интерфейса, параметр типа, псевдоним типа, константная переменная, свойство) и его атрибуты (является ли статическим членом класса, private членом класса, var-параметром).

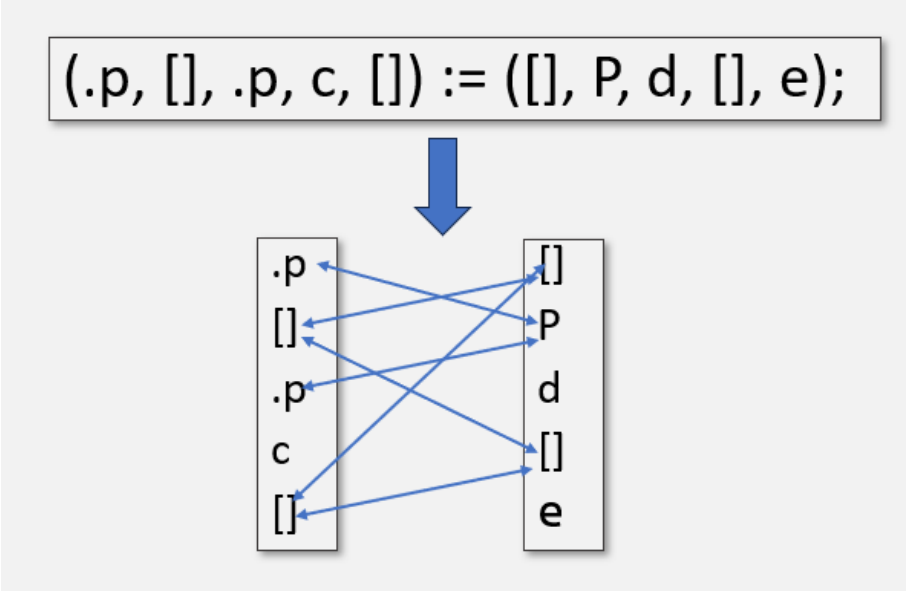
2. Алгоритм обработки различных типов символов

Как говорилось ранее, синонимами между собой могут быть составные имена и имена из внешнего контекста, у которых совпадает последнее имя; элементы массивов. Var-параметры могут оказаться одной ячейкой памяти с любым нелокальным именем.

Если возможные синонимы находятся в обоих частях кортежного присваивания, то без алгоритма их обработки возможно появление некорректных присваиваний, как было показано в Рис. 1. Если же возможные синонимы стоят только с одной стороны, то некорректные присваивания не возникают, так как значение символа либо только используется, либо только перезаписывается.

Визуально эту связь “возможно синонимы” можно представить графом, ребра которого соединяют возможные синонимы из разных частей кортежного присваивания, как показано на Рис 2. Для краткости далее используются следующие сокращения: .”имя” – составное имя, оканчивающиеся на “имя”; имя, написанное заглавными буквами – имя, пришедшее из внешнего контекста; [] – элемент массива; (var) – var-параметр; var t – временная переменная.

Рис. 2. Визуализация связи “возможно синонимы”



Так как некорректные присваивания невозможны в ситуации, когда возможные синонимы стоят только с одной стороны, то суть алгоритма заключается в том, чтобы оставить возможные синонимы только с одной стороны, минимизируя при этом количество дополнительных присваиваний. Визуально это значит, что на графе не должно остаться ребер.

Для достижения этой цели используются следующие преобразования:

1. Вынесение присваивания
2. Введение временной переменной

Вынесение присваивания возможно, если символ, стоящий напротив возможного синонима, отсутствует в противоположной ему части кортежного присваивания и не может быть ни с чем синонимом там же. Например, на Рис. 3 можно вынести последнее присваивание, так как `c` не присутствует с правой стороны, но в примере на Рис. 4 вынести последнее присваивание нельзя, так как `.p` может быть синонимом с чем-то справа.

Рис. 3. Присваивание можно вынести

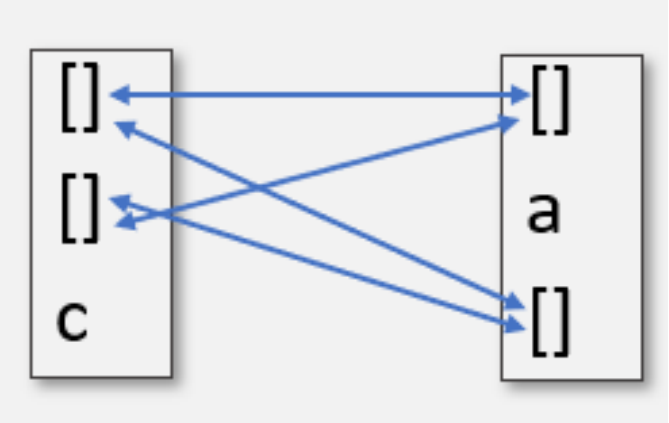
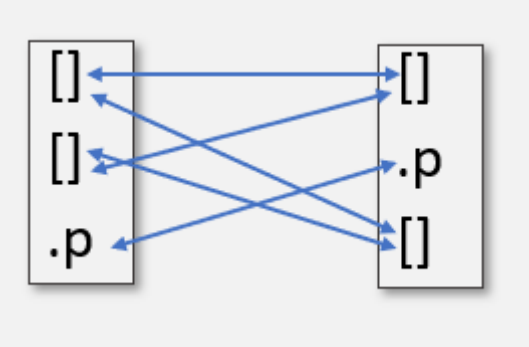


Рис. 4. Присваивание вынести нельзя



Если возможный синоним стоит справа, то присваивание выносится наверх (Рис. 5), если слева – вниз (Рис. 6).

Рис. 5. Вынос присваивания наверх

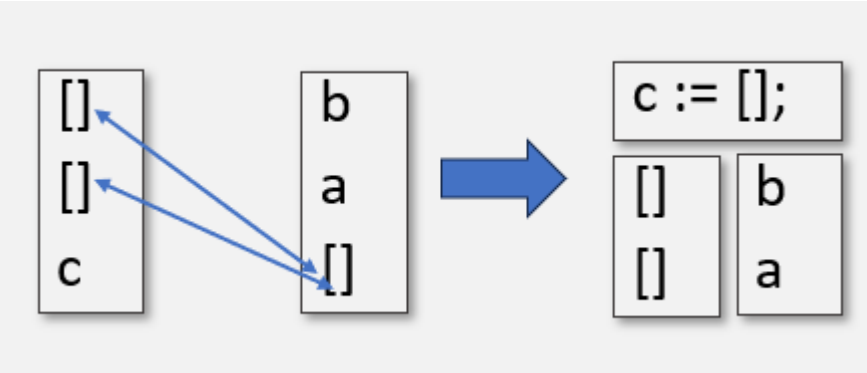
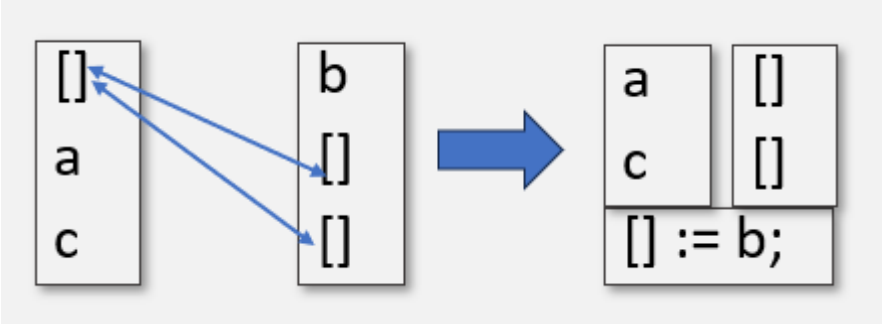
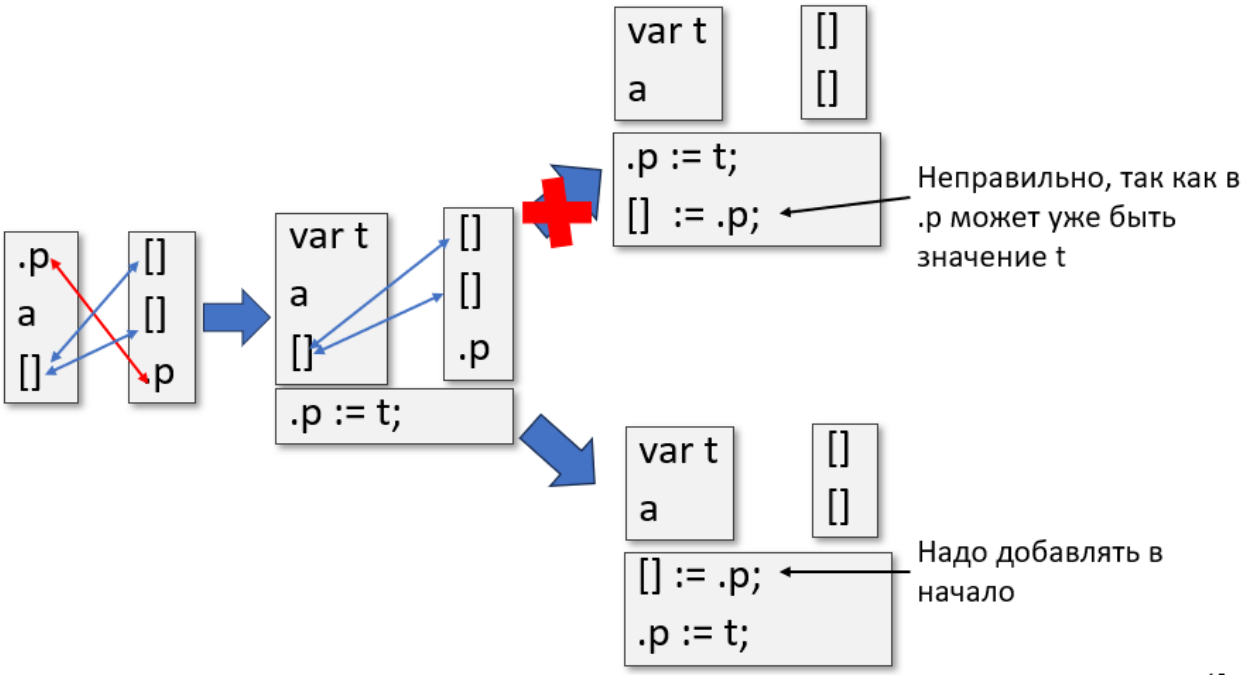


Рис. 6. Вынос присваивания вниз



При вынесении присваивания вниз, оно добавляется перед ранее вынесенными присваиваниями, а при вынесении вверх – после ранее вынесенных присваиваний. Если так не делать, то возможно появление некорректных присваиваний в случае присутствия нескольких графов в кортежном присваивании (Рис. 7).

Рис. 7. Появление некорректного присваивания при неправильном выносе присваивания



Если вынести присваивание невозможно, то вводится временная переменная. При этом появляется одно дополнительное присваивание, которое добавляется сверху, если возможный синоним был справа (Рис. 8), или снизу, если возможный синоним был слева (Рис. 9).

Рис. 8. Дополнительное присваивание сверху

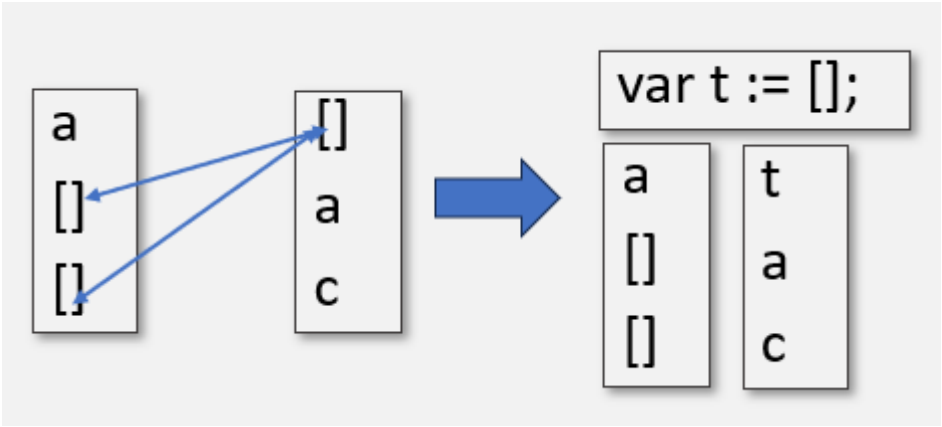
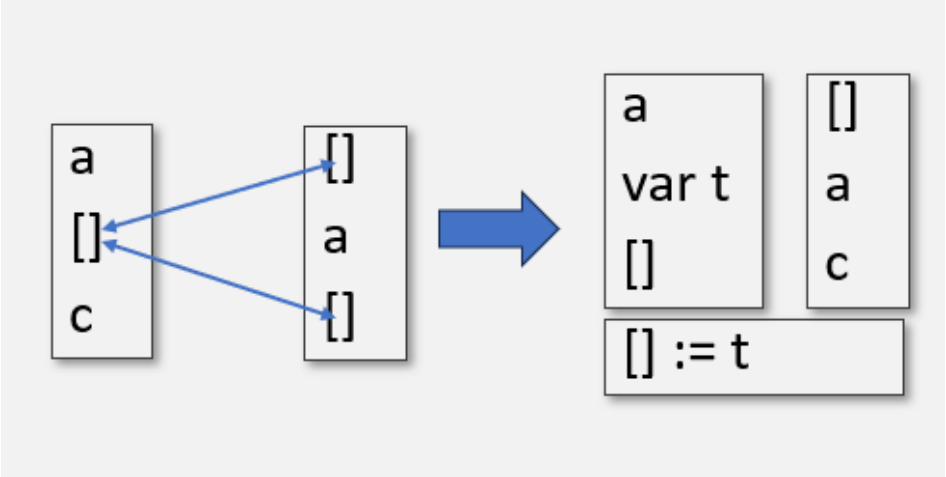


Рис. 9. Дополнительное присваивание снизу

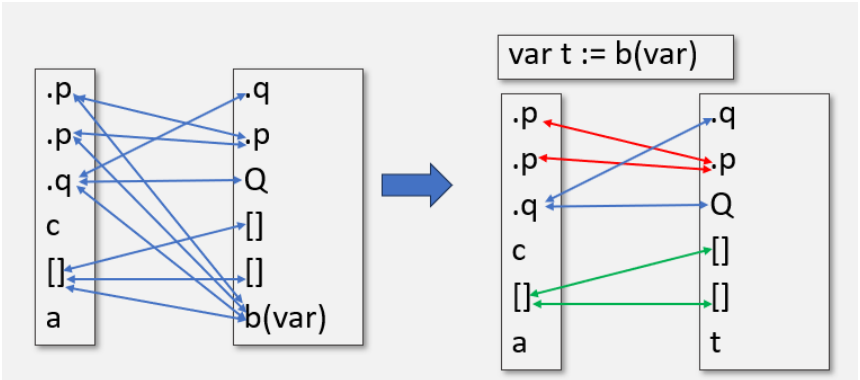


Дополнительные присваивания сверху должны идти перед вынесенными наверх присваиваниями, а дополнительные присваивания снизу должны происходить после вынесенных вниз присваиваний.

Так как var-параметры могут оказаться одной ячейкой памяти с любым нелокальным именем, для них всегда вводится временная переменная. Аналогично делается с выражениями, так как они могут содержать в себе символы из множественного присваивания.

После того как var-параметры вынесены из кортежного присваивания, графы возможных синонимов становятся в нем полносвязными в том смысле, что каждый возможный синоним одного типа (элемент массива, составные и нелокальные имена с совпадающим последним именем) с одной стороны связан с каждым возможным синонимом только того же типа с другой стороны (Рис. 10).

Рис. 10. Графы возможных синонимов после вынесения var-параметра

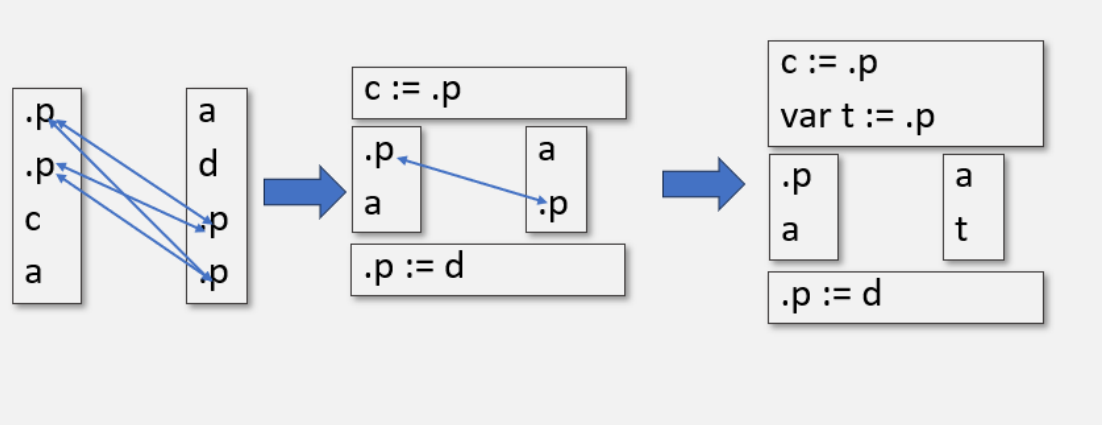


Теперь каждый из оставшихся графов нужно разрешить, то есть убрать в нем ребра, минимизируя при этом количество присваиваний и не генерируя некорректных присваиваний.

Алгоритм разрешения графа (Рис. 11):

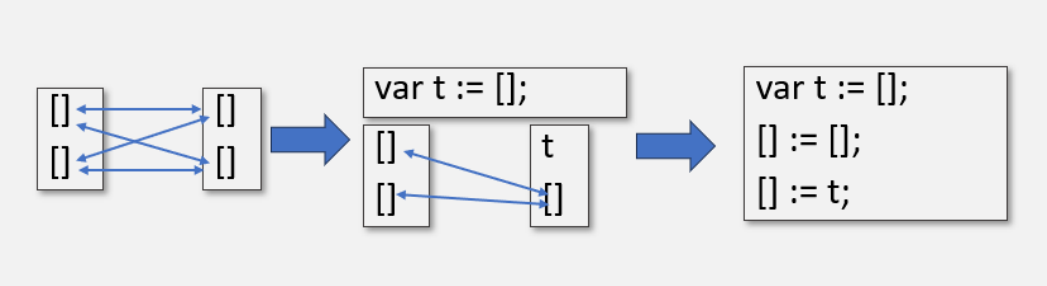
1. Вынести все присваивания, которые можно вынести на этот момент
2. Выбрать сторону, на которой осталось меньше возможных синонимов
3. Выносить присваивания или вводить новую переменную для возможных синонимов из выбранной стороны

Рис. 11. Визуализация алгоритма разрешения графа возможных синонимов



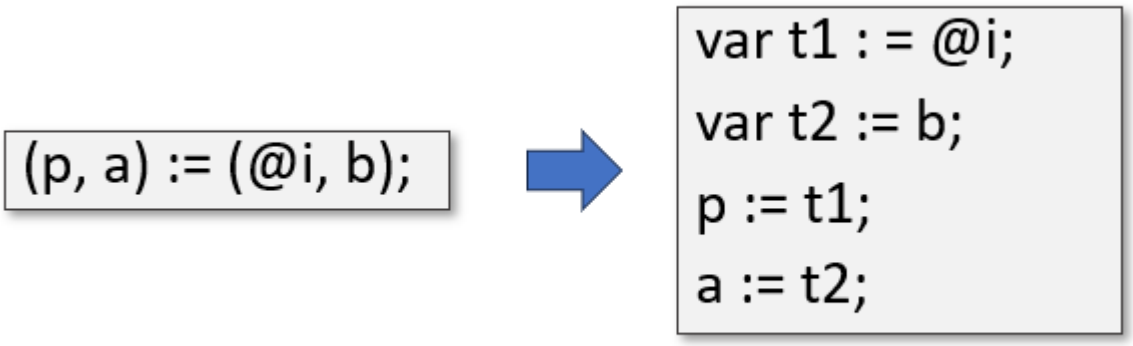
В третьем шаге алгоритма вынесение присваивания возможно, если в кортежном присваивании возможные синонимы одного типа стоят друг напротив друга, тогда одну из таких пар можно и нужно вынести (Рис. 12).

Рис. 12. Пример, когда можно вынести присваивание на третьем шаге алгоритма



Если в кортежном присваивании присутствуют операция разыменования или получения адреса, то оно не оптимизируется и раскрывается в $2 * n$ присваиваний, где n – количество элементов в левом кортеже. (Рис. 13).

Рис. 13. Обработка указателей



Таким образом, алгоритм обработки различных типов символов состоит из следующих шагов:

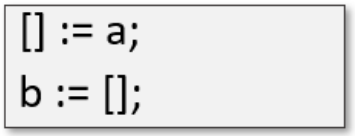
1. Если есть операция разыменования или получения адреса – сделать $2 * n$ присваиваний и закончить алгоритм
2. Ввести временные переменные для выражений и var-параметров
3. Разрешить все графы возможных синонимов

Утверждение 1.

В результате работы алгоритма не генерируется некорректных присваиваний.

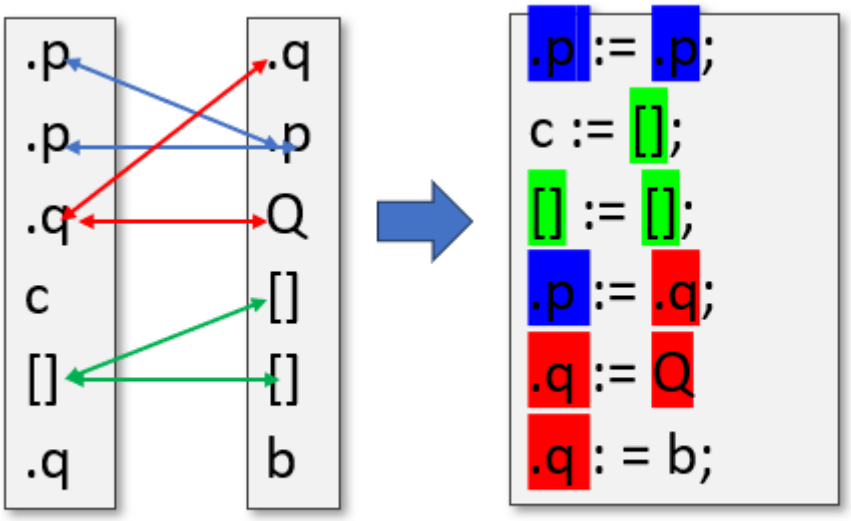
Для доказательства этого утверждения сформулируем признак возможного появления некорректного присваивания – некорректное присваивание может возникнуть только тогда, когда возможному синониму присваивается значение перед тем, как он используется (Рис. 14).

Рис. 14. Иллюстрация признака



Поэтому алгоритм должен располагать присваивания таким образом, чтобы сначала значения возможных синонимов одного типа присваивались в переменные, и только потом значения присваивались им (Рис. 15). На рисунке возможные синонимы одного типа выделены одним цветом.

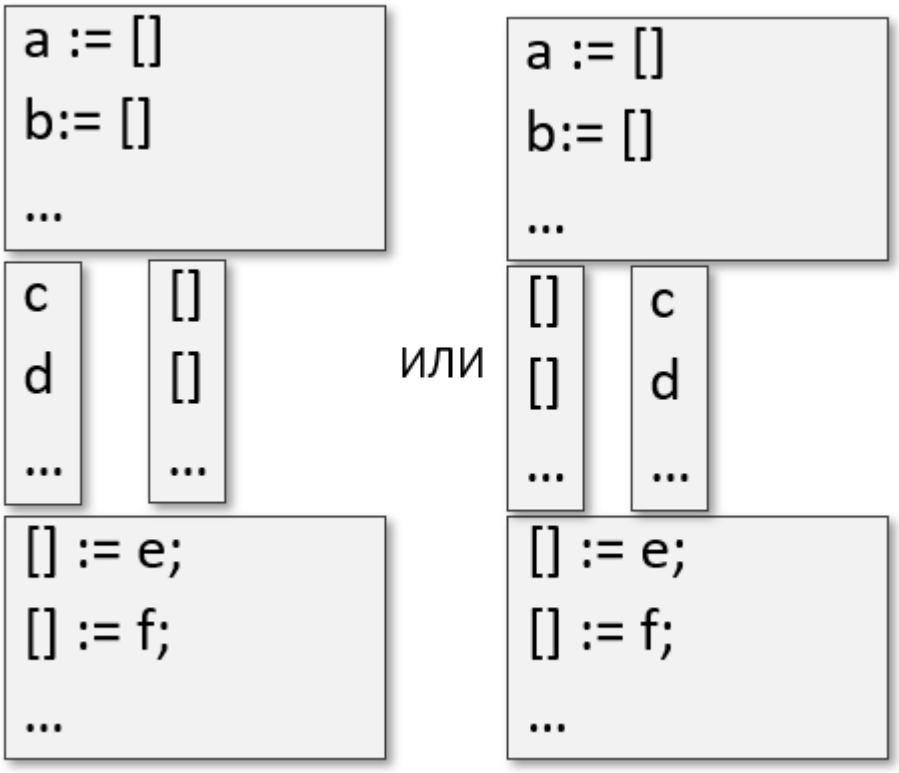
Рис. 15. Пример правильного раскрытия кортежного присваивания



Доказательство.

Очевидно, что для одного графа алгоритм работает правильно, так как он выносит все возможные синонимы либо с правой стороны наверх, либо с левой стороны вниз, исключая таким образом ситуацию, описанную в признаке. Схематично результат разрешения графа изображен на Рис. 16. Если при разрешении остальных графов в кортежном присваивании придется выносить присваивания, содержащие в себе символы из ранее разрешенных графов, то из Рис. 16 видно, что при правильном выносе (вверх – после ранее вынесенных, вниз – перед ранее вынесенными) некорректных присваиваний не происходит.

Рис. 16. Результат разрешения одного графа возможных присваиваний схематично



4. Изменения алгоритма раскрытия множественного присваивания

Согласно [3], алгоритм раскрытия множественного присваивания состоит из следующих шагов:

1. Удаление избыточных присваиваний
2. Обработка различных типов символов и выражений
3. Создание графа присваиваний
4. Поиск простых циклов графа
5. Устранение циклов графа
6. Обход леса и генерация отдельных присваиваний

При внедрении алгоритма в компилятор PascalABC.NET он был подвергнут следующим изменениям:

1. На первом шаге алгоритма избыточные присваивания не удаляются. Напомним, что к избыточным присваиваниям относятся самоприсваивания – имя присваивается само себе, и повторные присваивания – имя появляется в левом кортеже более одного раза. Самоприсваиввания выносятся. А если есть повторные присваивания, то генерируется $2 * n$ присваиваний.
2. Второй шаг алгоритма заменяется на описанный в третьем пункте алгоритм.
3. Алгоритм Джонсона[4] поиска всех простых циклов на графе заменяется на алгоритм проверки каждой компоненты связности на наличие цикла. Это делается потому, что граф присваиваний – псевдолес[5], а каждая компонента связности псевдолеса содержит не более одного цикла. Для поиска компонент связности используется алгоритм Косарайю[6].

5. Внедрение алгоритма в компилятор PascalABC.NET

Для внедрения алгоритма в компилятор PascalABC.NET был написан визитор NewAssignTuplesDesugarVisitor, который наследуется от визитора AssignTuplesDesugarVisitor, который использовался для раскрытия кортежных присваиваний для этого.

Если в множественном присваивании справа стоит кортеж, то визитор NewAssignTuplesDesugarVisitor использует описанный выше алгоритм оптимизации, в противном случае он вызывает реализацию предка (Листинг 5).

Листинг 5. Код раскрытия множественного присваивания.

```
public override void visit(assign_tuple node)
{
     if (node.expr is tuple_node tn)
     {
          var n = node.vars.variables.Count();
          if (n > tn.el.Count)
               throw new SyntaxVisitorError(
               "TOO_MANY_ELEMENTS_ON_LEFT_SIDE_OF_TUPLE_ASSIG” +
               “NMENT", node.vars.variables[0]);

          var assigns = desugar(tn, node.vars);
          ReplaceStatementUsingParent(node, assigns);
     }
     else
          base.visit(node);
}
```

Еще визитор NewAssignTuplesDesugarVisitor оптимизирует кортежные объявления, то есть конструкции вида `var (a, b) := (c ,d)`, переводя их в n одиночных присваиваний (Листинг 6).

Листинг 6. Код оптимизации кортежных объявлений

```
public override void visit(assign_var_tuple node)
{
     if (node.expr is tuple_node tn)
     {
          var n = node.idents.idents.Count();
          if (n > tn.el.Count)
```

```
                throw new SyntaxVisitorError("TOO_MANY_ELE-
            MENTS_ON_LEFT_SIDE_OF_TUPLE_ASSIGNMENT",
            node.idents.idents[0]);

            var assigns = new List<statement>();
            for(var i =0; i < n; i++)
            {
                var cur =
                new var_def_statement(node.idents.idents[i],
                tn.el.expressions[i], tn.Parent.source_context);

                assigns.Add(new var_statement(cur,
                node.source_context));
            }
            ReplaceStatementUsingParent(node, assigns);
        }
        else
            base.visit(node);

}
```

6. Сравнения количества присваиваний и производительности

Были проведены сравнения количества генерируемых присваиваний с компилятором C# (Рис. 17 - 20). Для просмотра сгенерированного IL-кода использовалась программа ILSpy.

Рис. 17. Сравнение для `(a, b) = (b,a)`

```
(a, b) := (b, a);
```

C#

```
int num = b;
b = a;
a = num;
```

PascalABC.NET

```
int num3 = num;
num = num2;
num2 = num3;
```

Рис. 18. Сравнение для `(a, b, c, d, e, f) = (b, c, d, e, f, a)`

```
(a, b, c, d, e, f) := (b, c, d, e, f, a);
```

C#

```
int num = b;
int num2 = c;
int num3 = d;
int num4 = e;
int num5 = f;
f = a;
e = num5;
d = num4;
c = num3;
b = num2;
a = num;
```

PascalABC.NET

```
int num7 = num3;
num3 = num4;
num4 = num5;
num5 = num6;
num6 = num;
num = num2;
num2 = num7;
```

Рис. 19. Сравнение для `([], []) = ([], [])`

```
([], []) := ([], [])
```

C#

```
ref int reference = ref arr[i];
ref int reference2 = ref arr[j];
int num = arr[j];
int num2 = arr[i];
reference = num;
reference2 = num2;
```

PascalABC.NET

```
int num10 = array[num7];
array[num7] = array[num8];
array[num8] = num10;
```

Рис. 20. Сравнение для `(.p, .p, .p)=(.p,.p,.p)`

```
(.p, .p, .p) = (.p, .p, .p)
```

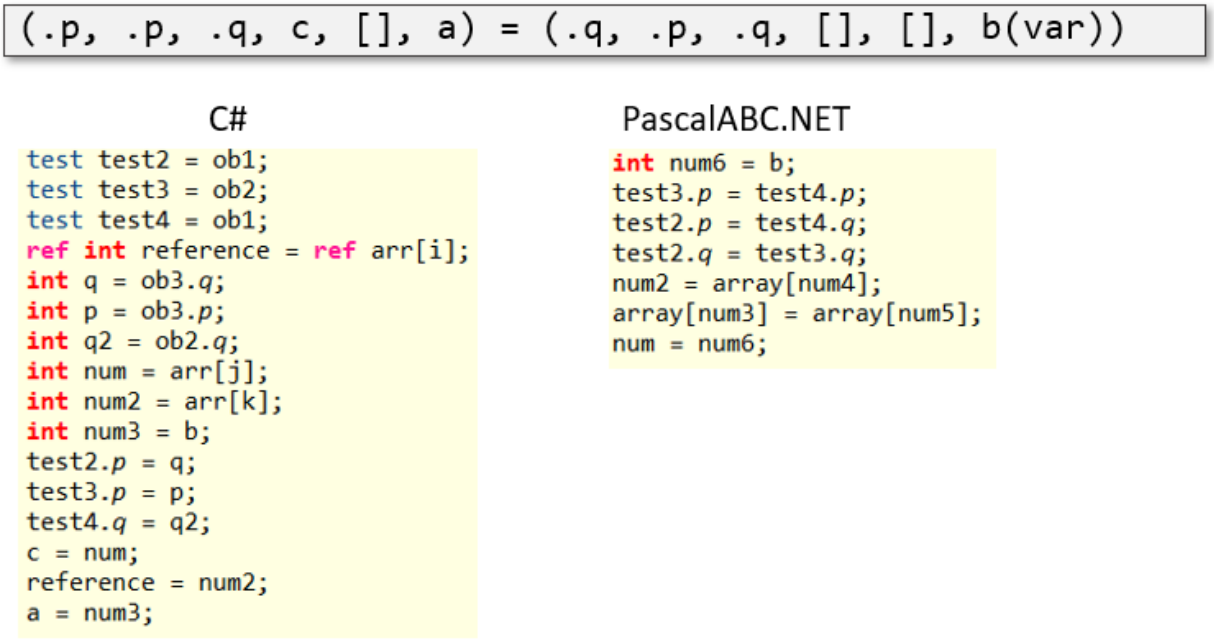
C#

```
test test2 = ob2
test test3 = ob3
int p = ob2.p;
int p2 = ob3.p;
int p3 = ob1.p;
ob1.p = p;
test2.p = p2;
test3.p = p3;
```

PascalABC.NET

```
int p = test4.p;
int p2 = test2.p;
test2.p = test3.p;
test3.p = p;
test4.p = p2;
```

Рис. 21. Сравнение для `(.p, .p, .q, c, [], a) = (.q, .p, .q, [], [], b(var))`



Результаты сравнения приведены в Таблице 1.

Таблица 1. Результаты сравнения

Раскрываемое множественное присваивание	Количество отдельных присваиваний	
	C#	PascalABC.NET
`(a, b) = (b, a)`	3	3
`(a, b, c, d, e, f) = (b, c, d, e, f, a)`	11	7
`([], []) = ([], [])`	6	3
`(.p, .p, .p) = (.p,.p,.p)`	8	5
`(.p, .p, .q, c, [], a) = (.q, .p, .q, [], [], b(var))`	16	7

Из результатов сравнения можно сделать вывод, что компилятор PascalABC.NET делает меньше или столько же присваиваний. Причем в примерах, где присутствуют нелокальные символы, компилятор C# делает вплоть до

$3 * n$ присваиваний, в то время как компилятор PascalABC.NET максимум генерирует $2 * n$ присваиваний.

Однако, в некоторых ситуациях делать $3 * n$ присваиваний необходимо, иначе возможно неправильное раскрытие кортежного присваивания (Рис.22). Такое случается, если одно имя является частью другого (`a` и `a[i]`).

Рис. 22. Пример неправильного раскрытия кортежного присваивания

```
(a, a[i]) = (b, c)
```

C#

```
ref int reference = ref arr[i];
int num = c;
arr = arr2;
reference = num;
```

PascalABC.NET

```
array = array2;
array[num7] = num3;
```

PacsalABC.NET раскрывает такие примеры неправильно, так как для правильного их раскрытия нужны ref-конструкция и функционал, позволяющий находить имена, являющиеся частью других. Отметим, что Python тоже работает в таких случаях неправильно (Рис. 23).

Рис. 23. Неправильное раскрытие кортежного присваивания в языке Python

Код

```
a = [1, 2, 3]
c = a
b = [4, 5, 6]
(a, a[0]) = (b, 10)
print(b[0])
print(a[0])
print(c[0])
```

Вывод

```
10
10
1
```

Также был измерен прирост производительности при использовании нового алгоритма. Для этого измерялось среднее время работы двух алгоритмов, использующих кортежное присваивание. (Листинг 7 и Листинг 8). Результаты измерений представлены в Таблице 2.

Листинг 7. Первый алгоритм

```
function measureSort(n : integer) : Integer;
begin
  var a := ArrRandomInteger(n,0,MaxInt-1);
  MillisecondsDelta;
  for var i := 1 to n-1 do
        for var j := 1 to n-1-i do
            if a[j] > a[j+1] then begin
                (a[j], a[j+1]) := (a[j+1], a[j]);
            end;
  Result := MillisecondsDelta;
end;
```

Листинг 8. Второй алгоритм

```
function measure2(n : integer) : Integer;
begin
  var (a, b, c, d, e) := (1, 2, 3, 4, 5);
  MillisecondsDelta;
  loop n do
      (a, b, c, d, e) := (b, c, d, e, a);
  Result := MillisecondsDelta;
end;
```

Таблица 2. Замеры производительности

Алгоритм	Без оптимизации	С оптимизацией	Прирост производительности
1	320 мс	280 мс	12.5%
2	3.1 мс	2.1 мс	32%

Заключение

В результате работы были получены следующие результаты

1. Написан визитор для поиска информации о символах
2. Разработан алгоритм оптимизации множественного присваивания для различных типов символов
3. Разработанный алгоритм внедрен в компилятор PascalABC.NET
4. Проведено сравнение количества генерируемых присваиваний с компилятором C#
5. Показан прирост производительности при использовании оптимизации

Код проекта на GitHub (ветка opt_tuple_assignment) – https://github.com/pascalabcnet/pascalabcnet

Литература

1. Кортежное присваивание в C# – URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples#tuple-assignment-and-deconstruction (дата обращения 1.06.2024)
2. Кортежное присваивание в Python – URL: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences
3. Филонов А. К. Оптимизация множественного присваивания в PascalABC.NET – URL: https://hub.sfedu.ru/repository/material/801312059/ (дата обращения 1.06.2024)
4. Donald B. Johnson 1975. Finding All the Elementary Circuits of a Directed Graph. SIAM J. Comput., 4(1), p.77–84.– Алгоритм Джонсона поиска всех простых циклов на графе.
5. Kruskal, C., Rudolph, L., and Snir, M. 1990. Efficient Parallel Algorithms for Graph Problems. Algorithmica, 5, p.43-64. – Псевдолес
6. https://ru.wikipedia.org/wiki/Алгоритм_Косарайю - Алгоритм Косарайю