Journal of Siberian Federal University. Engineering & Technologies 2020 13(7): 777–787
~ ~ ~

DOI: 10.17516/1999-494X-0230

УДК 004.416.2

Development of Procedural-Parametric Paradigm in the Language GO

Sergey Yu. Smogluk*,
Eugeniy N. Garin and Daria S. Romanova
Siberian Federal University
Krasnoyarsk, Russian Federation

Received 11.02.2020, received in revised form 18.02.2020, accepted 14.03.2020

Abstract. The article presents a programming paradigm that defines a new style of program development called procedural-parametric programming (PPP). The paradigm is based on parametric polymorphism, which allows the procedures to accept and process variant data types without the algorithmic choice of alternatives within these procedures. In procedural programming languages, such types are described by unions (union in C, C++) or variant entries (in Pascal). Algorithmic processing of variants is carried out by means of conditional operators or switches. This approach is a development of procedural programming methods and acts as an alternative to object-oriented programming. The procedural-parametric paradigm of programming is an extension of the procedural approach. It makes possible to increase the capabilities of the latter by supporting data polymorphism. The application of the proposed approach will allow to increase the functional capabilities of the procedures without making any internal algorithmic changes. Procedural-parametric programming can be used both independently and in combination with other programming paradigms.

Keywords: procedural-parametric paradigm, parametric polymorphism, structural programming, procedural programming, data processing, programming languages, data polymorphism, algorithm changes.

Citation: Smogluk S.Yu., Garin E.N., Romanova D.S. Development of procedural-parametric paradigm in the language GO, J. Sib. Fed. Univ. Eng. & Technol., 2020, 13(7), 777–787. DOI: 10.17516/1999-494X-0230

* Corresponding author E-mail address: xok-s@yandex.ru

Разработка процедурно-параметрической парадигмы на языке GO

С.Ю. Смоглюк, Е.Н. Гарин, Д.С. Романова
Сибирский федеральный университет
Российская Федерация, Красноярск

Аннотация. В статье рассматривается парадигма программирования, определяющая новый стиль разработки программ, названный процедурно-параметрическим программированием (ППП). В основе парадигмы лежит параметрический полиморфизм, позволяющий процедурам принимать и обрабатывать вариантные типы данных без алгоритмического выбора альтернатив внутри этих процедур. В процедурных языках программирования такие типы описываются объединениями (union в языках C, C++) или вариантными записями (в языке Паскаль). Алгоритмическая обработка вариантов осуществляется с применением условных операторов или переключателей. Данный подход является развитием методов процедурного программирования и служит альтернативой объектно-ориентированному программированию. Процедурно-параметрическая парадигма программирования является расширением процедурного подхода. Она позволяет увеличить возможности последнего за счет поддержки полиморфизма данных. Применение предлагаемого подхода позволит наращивать функциональные возможности процедур без внесения в них внутренних алгоритмических изменений. ППП может использоваться как независимо, так и в сочетании с другими парадигмами программирования.

Ключевые слова: процедурно-параметрическая парадигма, параметрический полиморфизм, структурное программирование, процедурное программирование, обработка данных, языки программирования, полиморфизм данных, алгоритмические изменения.

Цитирование: Смоглюк, С.Ю. Разработка процедурно-параметрической парадигмы на языке GO / С.Ю. Смоглюк, Е.Н. Гарин, Д.С. Романова // Журн. Сиб. федер. ун-та. Техника и технологии, 2020. 13(7). С. 777–787. DOI: 10.17516/1999-494X-0230

Введение

Предлагается парадигма, определяющая новый стиль разработки программ, названный процедурно-параметрическим программированием (ППП). В ее основе лежит параметрический полиморфизм, позволяющий процедурам принимать и обрабатывать вариантные типы данных без дополнительного алгоритмического анализа. Подход является развитием методов процедурного программирования и может служить альтернативой объектно-ориентированному стилю [1-9].

У Страуструпа он описан при демонстрации указателей на функции. Ясно также, что непосредственное использование таких приемов не повысит шансов процедурного программирования в борьбе с ОО-стилем. Ведь поиск и обнаружение ошибок в программе необходимо осуществлять во время ее выполнения. Однако приведенный пример указывает путь по дальнейшему языковому расширению процедурного подхода.

Различие процедурной и объектной парадигмы

Различия существующих парадигм программирования заключаются в их особенностях обеспечения поддерживать современные методологии разработки программного обеспечения.

Основными требованиями к разработке программного обеспечения являются: удобство сопровождения, возможность наращивания уже существующей функциональности, способность использования существующего кода к повторному использованию. При этом на второй план отступает быстрое проектирование первоначальной версии программы, так как его воплощение обычно не позволяет соблюсти все остальные условия.

Предъявляемым требованиям соответствует объектно-ориентированное программирование (ООП). Развитие ООП практически полностью вытеснило процедурное программирование из области, связанной с разработкой сложных программных систем.

Основной конструктивный элемент ООП – класс. Именно он определяет свойства, связанные с возможностью однократного определения типа обрабатываемого объекта и последующим многократным использованием его внутренних процедур для изменения своих данных без алгоритмической проверки типа. Эта группировка осуществляется на основе построения однозначных отношений между специализированными данными и обрабатывающими их процедурами. При процедурном подходе такая группировка обычно осуществляется внутри процедур, что ведет к использованию алгоритмических методов для ее формирования и обработки. ООП предлагает группировку процедур вокруг обрабатываемых ими альтернативных наборов данных, что позволяет обойтись только декларативными методами.

Поэтому в рамках данного исследования в работе изучена возможность и эффективность использования новой парадигмы программирования, основанной на процедурно-параметрических массивах (ППП) в новом языке программирования от *Google – GO*.

Процедурно-параметрическая парадигма

Существует еще один вариант парадигмы. Мы можем предварительно установить новую функциональную зависимость $F(X)$, как $G(f_i, tj)$ между процедурой (f_i) и данными (t_j). Такая однозначная зависимость называется параметрической. Использование параметрической зависимости между аргументами позволяет убрать иерархию из анализа вариантов и обеспечивает независимость одного аргумента от другого, в данном случае процедур от данных. Такой подход составляет основу параметрического полиморфизма и процедурно-параметрической парадигмы программирования.

Моделирование параметрического выбора альтернатив на языке GO

Одним из простых приемов параметризации выступает применение многомерных массивов, в которых каждая из размерностей определяет один из варьируемых параметров. Значениями элементов такого массива являются выбираемые альтернативы. Для рассматриваемого примера будем строить двухмерный массив, или таблицу (табл. 1).

Для табличного доступа не имеет значения, в какой последовательности определяются и используются индексы элемента. Программная реализация выбора альтернатив с применением параметров позволяет получить еще один подход, который можно назвать процедурно-параметрическим программированием.

Параметрический подход можно смоделировать, используя процедурное программирование. Ниже представлена программа нахождения площади и периметра фигуры, использу-

ющая параметрические отношения между альтернативными типами данных и функций. Для того чтобы иметь возможность обрабатывать значения типов без использования механизма их идентификации во время выполнения, введем перечислимый тип данных, каждое значение которого соответствует одному из типов (табл. 2).

Параметризации подвергаются два вида функции программы: функции расчета площади (*Square*), функции расчета периметра фигуры (*Perimeter*) и вывода фигур (*Out*).

Таблица 3 описывает параметризируемые функции данной программы.

Язык программирования GO позволяет создавать модули, благодаря которым код программы можно разбить на разные каталоги для дальнейшего расширения функциональности (рис. 1), а каталоги разбить на процедуры и типы данных (рис. 2).

Благодаря такой архитектуре с помощью языка GO можно написать код, разбитый по файлам и представленный в листинге.

Таблица 1. Зависимость параметрического массива от данных

Table 1. Parametric array data dependency

Значение функции F	Значение типа данных T			
	T_1	T_2	...	t_n
F_1	F_{11}	F_{12}	...	F_{1n}
F_2	F_{12}	F_{22}	...	F_{2n}
...	...	...	...	...
f_m	f_{m1}	f_{m2}	...	f_{mn}

Таблица 2. Перечисляемые типы данных

Table 2. Enumerated Data Types

Вид фигуры	Обозначение типа данных	Значение перечислимого типа
Прямоугольник	*Rectangle*	*RECTANGLE*
Треугольник	*Triangle*	*TRIANGLE*
Круг	*Circl*	*CIRCL*

Таблица 3. Описание параметризируемых функций

Table 3. Description of parameterized functions

Значение функции F	Значение типа данных T			
	RECTANGLE	*TRIANGLE*	*CIRCL*	t_n
Square	*Square_rectangle*	*Square_triangle*	*Square_circl*	F_{1n}
Perimeter	*Perimeter_rectangle*	*Perimeter_triangle*	*Perimeter_ circl*	F_{2n}
Out	*OutRectangle*	*OutTriangle*	*OutCircl*	...
f_m	f_{m1}	f_{m2}	...	f_{mn}

Рис. 1. Пакетная модульность программы

Fig. 1. Batch modularity of the program

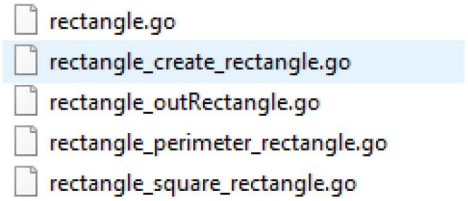

Рис. 2. Модульность программы

Fig. 2. Program modularity

Листинг-моделирование
параметрического полиморфизма на языке GO

Реализация инициализации полиморфных объектов типа Shape изображена на рис. 3. Явно видно, что три (*rec, tri, cir*) объекта представляют собой один общий тип данных, как абстрактный класс в ООП, но в каждом объекте хранится совсем другой тип данных, более уточненный.

На рис. 4 представлен код, реализующий обобщенный тип данных Shape, который является обобщением фигур треугольника, круга и прямоугольника. Реализованы процедуры динамического создания обобщенного круга.

На рис. 5, 6 и 7 изображен код, реализующий процедуры динамического создания обобщенного прямоугольника, треугольника и уточнение специализаций с помощью параматрических массивов, хранящих в себе процедуры, вычисляющие площадь и периметр определенной фигуры.

На рис. 5 и 6 представлен код, реализующий процедуры динамического создания обобщенного прямоугольника, треугольника и уточнение специализаций с помощью параматрических массивов, хранящих в себе процедуры, вычисляющие площадь и периметр определенной фигуры.

Каталог “*circle*”, файл “*circle.go*”:

```
package circle
type Circl struct {
   r float64
}
```

Каталог “*circle*”, файл “*circle_create_circlt.go*”:

```
package circle
//-----------------------------------------------------------------
```

```
package main

import (
    "fmt"
    "../shape"
)

//---------------------------------------------------------------------
//Главная функция Создает 3 объекта обобщения:
// Прямоугольник, треугольник, круг
// 1) Выводи на экран созданные специализации
// Out(...) -- Обобщенная процедура возвращает имя специализации созданные чере обобщение
// 2) Выводит на экран расчитанную площадь каждой специализации
// Square(...) -- Обобщенная процедура расчитывающая чере обобщение площадь специализации
// 2) Выводит на экран расчитанную площадь каждой специализации
// Perimeter(...) -- Обобщенная процедура расчитывающая чере обобщение периметр специализации
func main() {

    rec := shape.Create_shape_rectangle(1,2)
    tri := shape.Create_shape_triangle(1,3,3)
    cir := shape.Create_shape_circl(4)

    fmt.Printf("\n=====Create type data=====\n")
    fmt.Printf("\n%s, %s, %s\n",shape.Out(rec),shape.Out(tri),shape.Out(cir))

    fmt.Printf("\n=====Next Square=====\n")

    shape.Square(rec);
    shape.Square(tri);
    shape.Square(cir);

    fmt.Printf("\n=====Next Perimetr=====\n")

    shape.Perimeter(rec);
    shape.Perimeter(tri);
    shape.Perimeter(cir);

    fmt.Printf("\n=====Exit=====\n")
}
```

Рис. 3. Каталог “main”, файл “main.go” (1)

Fig. 3. The “main” directory, the “main.go” file (1)

// Инициализация круга

*func Create_circl(r float64) *Circl {*

item := new(Circl)

item.r = r

return item

}

Каталог “*circle*”, файл “*circle_outCircl.go*”:

package circle

func OutCircl() string {

return “Circl”

}

Каталог “*circle*”, файл “*circle_perimeter_circl.go*”:

package circle

*func Perimeter_circl(item *Circl) float64 {*

*perimeter := 2 * 3.14 * item.r*

return perimeter

}

```
package shape
import (
    "fmt"

    "../circle"
    "../rectangle"
    "../triangle"
)
//---------------------------------------------------------------------
//Enum перечисление
// значения локальных ключей для каждой из фигур
type key int
const (
    RECTANGLE key = 0
    TRIANGLE      = 1
    CIRCL         = 2
)
//---------------------------------------------------------------------
//Указатель на любой тип данных
// используется универсальный указатель
type Figure interface {
}
//---------------------------------------------------------------------
// Использование параметрического обобщения, построенного на
// основе косвенного альтернативного связывания специализаций.
// Структура, обобщающая все имеющиеся фигуры
type Shape struct {
    k   key    // ключ
    ptr Figure // подключается любая специализация
}
//---------------------------------------------------------------------
// Динамическое создание обобщенного прямоугольника
func Create_shape_rectangle(a, b float64) *Shape {
    s := new(Shape)
    s.k = RECTANGLE
    s.ptr = rectangle.Create_rectangle(a, b)
    return s
}
```

Рис. 4. Каталог “shape”, файл “shape.go” (2)

Fig. 4. The “shape” directory, the “shape.go” file (2)

```
//---------------------------------------------------------------------
// Динамическое создание обобщенного треугольника
func Create_shape_triangle(a, b, c float64) *Shape {
    s := new(Shape)
    s.k = TRIANGLE
    s.ptr = triangle.Create_triangle(a, b, c)
    return s
}
//---------------------------------------------------------------------
// Динамическое создание обобщенного круга
func Create_shape_circl(r float64) *Shape {
    s := new(Shape)
    s.k = CIRCL
    s.ptr = circle.Create_circl(r)
    return s
}
var listPointOut []func() string
//---------------------------------------------------------------------
// Дополнительное переопределение процедуры вывода какая фигу используется
// обобщенной фигуры, сделанное для сокрытия ее реального вида.
// Может отсутствовать.
func Out(fp *Shape) string {
    listPointOut = []func() string{rectangle.OutRectangle, triangle.OutTriangle}
    listPointOut = append(listPointOut, circle.OutCircl)
    ans := listPointOut[fp.k]()
    return ans
}
//---------------------------------------------------------------------
// Дополнительное переопределение процедуры вычисления площади
// обобщенной фигуры, сделанное для сокрытия ее реального вида.
// Может отсутствовать.
func Square(item *Shape) {
    listPoint := []func(item *Shape) float64{square_rectangle_of_shape, square_triangle_of_shape, square_circl_of_shape}
    ans := listPoint[item.k](item)
    name := Out(item)
    fmt.Printf("\nSquare %s = %f\n", name, ans)
}
```

Рис. 5. Каталог “shape”, файл “shape.go” (3)

Fig. 5. The “shape” directory, the “shape.go” file (3)

```
//---------------------------------------------------------------------
// Дополнительное переопределение процедуры вычисления периметра
// обобщенной фигуры, сделанное для сокрытия ее реального вида.
// Может отсутствовать.
func Perimeter(item *Shape) {
    listPoint := []func(item *Shape) float64{perimeter_rectangle_of_shape, perimeter_triangle_of_shape, perimeter_circl_of_shape}
    ans := listPoint[item.k](item)
    name := Out(item)
    fmt.Printf("\nPerimeter %s = %f\n", name, ans)
}
//---------------------------------------------------------------------
// Обработчикики специализаций
// предназначенный для вычисления площади
// прямоугольника. Используется как элемент параметрического массива
// в процедуре вычисления площади обобщенной фигуры.
func square_rectangle_of_shape(item *Shape) float64 {
    return rectangle.Square_rectangle(item.ptr.(*rectangle.Rectangle))
}
//предназначенный для вычисления площади
// треугольника. Используется как элемент параметрического массива
// в процедуре вычисления площади обобщенной фигуры.
func square_triangle_of_shape(item *Shape) float64 {
    return triangle.Square_triangle(item.ptr.(*triangle.Triangle))
}
//предназначенный для вычисления площади
// круга. Используется как элемент параметрического массива
// в процедуре вычисления площади обобщенной фигуры.
func square_circl_of_shape(item *Shape) float64 {
    return circle.Square_circl(item.ptr.(*circle.Circl))
}
//---------------------------------------------------------------------
// Обработчикики специализаций
// предназначенный для вычисления периметра
// прямоугольника. Используется как элемент параметрического массива
// в процедуре вычисления периметра обобщенной фигуры.
func perimeter_rectangle_of_shape(item *Shape) float64 {
    return rectangle.Perimeter_rectangle(item.ptr.(*rectangle.Rectangle))
}
```

Рис. 6. Каталог “shape”, файл “shape.go” (4)

Fig. 6. The “shape” directory, the “shape.go” file (4)

```
// предназначенный для вычисления периметра
// треугольника. Используется как элемент параметрического массива
// в процедуре вычисления периметра обобщенной фигуры.
func perimeter_triangle_of_shape(item *Shape) float64 {
    return triangle.Perimeter_triangle(item.ptr.(*triangle.Triangle))
}
// предназначенный для вычисления периметра
// круга. Используется как элемент параметрического массива
// в процедуре вычисления периметра обобщенной фигуры.
func perimeter_circl_of_shape(item *Shape) float64 {
    return circle.Perimeter_circl(item.ptr.(*circle.Circl))
}
```

Рис. 7. Каталог “shape”, файл “shape.go” (5)

Fig. 7. The “shape” directory, the “shape.go” file (5)

Каталог “*circle*”, файл “*circle_square_circl.go*”:

```
package circle
func Square_circl(item *Circl) float64 {
   s := 3.14 * item.r
return s
}
```

Каталог “*rectangle*”, файл “*rectangle.go*”:

```
package rectangle
type Rectangle struct {
   a float64
b float64
}
```

Каталог “*rectangle*”, файл “*rectangle_create_rectangle.go*”:

```
package rectangle
//-------------------------------------------------------------------
// Инициализация прямоугольника
func Create_rectangle(a, b float64) *Rectangle {
   item := new(Rectangle)
   item.a = a
   item.b = b
   return item
}
```

Каталог “*rectangle*”, файл “*rectangle_outRectangle.go*”:

```
package rectangle
func OutRectangle() string {
   return “Rectangle”
}
```

Каталог “*rectangle*”, файл “*rectangle_perimeter_rectangle.go*”:

```
package rectangle
func Perimeter_rectangle(item *Rectangle) float64 {
   perimeter := 2 * (item.a * item.b)
   return perimeter
}
```

Каталог “*rectangle*”, файл “*rectangle_square_rectangle.go*”:

```
package rectangle
func Square_rectangle(item *Rectangle) float64 {
   s := item.a * item.b
   return s
}
```

Каталог ”*triangle*”, файл “*triangle.go*”:

```
package triangle
import (
   _ “fmt”
)
type Triangle struct {
   a float64
   b float64
   c float64
}
```

Каталог ”triangle”, файл “triangle_create_triangle.go”:

```
package triangle
//-------------------------------------------------------------------
// Инициализация треугольника
func Create_triangle(a, b, c float64) *Triangle {
```

```
    item := new(Triangle)
    item.a = a
    item.b = b
    item.c = c
    return item
}
```

Каталог ”*triangle*”, файл “*triangle_outTriangle.go*”:

```
package triangle
func OutTriangle() string {
    return “Triangle”
}
```

Каталог ”*triangle*”, файл “*triangle_perimeter_triangle.go*”:

```
package triangle
func Perimeter_triangle(item *Triangle) float64 {
    perimeter := item.a + item.b + item.c
    return perimeter
}
```

Каталог ”*triangle*”, файл “*triangle_square_triangle.go*”:

```
package triangle
import “math”
func Square_triangle(item *Triangle) float64 {
    p := float64(item.a+item.b+item.c) / 2.0
    s := math.Sqrt(p * (p – item.a) * (p – item.b) * (p – item.c))
    //fmt.Println(“p := float64(item.a + item.b + item.c)/2.0 = “, p)
    //fmt.Println(“s:= math.Sqrt(p*(p-item.a)*(p-item.b)*(p-item.c)) = “, s)
    return s
}
```

Заключение

Язык программирования GO дал возможность программисту разбить программу на пакетные модули, которые в свою очередь позволяют разбить код программы на отдельные файлы, что упрощает нахождение нужной процедуры для дальнейшего расширения функциональности.

Процедурно-параметрический подход позволяет использовать функциональный стиль программирования для реализации сложных высоконагруженных программ, а расширение функциональности удобней, чем при ООП-парадигме.

Благодарности / Acknowledgements

Работа выполнена при финансовой поддержке Министерства науки и высшего образования Российской Федерации в ходе реализации комплексного проекта «Создание высокотехнологичного производства земных станций перспективных систем спутниковой связи для обеспечения связанности труднодоступных, северных и Арктических территорий Российской

Федерации», осуществляемого при участии Сибирского федерального университета (соглашение № 075-11-2019-078 от 13/12/2019).

This work was financially supported by the Ministry of Science and Higher Education of the Russian Federation in the implementation of the integrated project «Creation of a production of earth stations of advanced satellite communications systems to ensure the coherence of hard, northern and Arctic territory of Russian Federation», implemented with the participation of the Siberian Federal University (agreement number 075 -11-2019-078 dated 13/12/2019).

Список литературы / References

[1] Легалов А.И. Функциональный язык для создания архитектурно независимых параллельных программ. *Вычислительные технологии.* 2005, 1(10), 71–89 [Legalov A.I. Functional language for creating architecture-independent parallel programs. *Computational technologies.* 2005, 1(10), 71–89 (in Russian)]

[2] Легалов А.И., Казаков Ф.А., Кузьмин Д.А., Привалихин Д.В. Модель функционально-потоковых параллельных вычислений и язык программирования «Пифагор». *Распределенные и кластерные вычисления. Избранные материалы второй Школы-семинара. Институт вычислительного моделирования СО РАН.* Красноярск, 2002, с. 101-120 [Legalov A.I., Kazakov F.A., Kuzmin D.A., Privalikhin D.V. The model of functional-stream parallel computing and the Pythagoras programming language. *Distributed and cluster computing. Selected materials of the second School-seminar. Institute of Computational Modeling SB RAS.* Krasnoyarsk, 2002, p. 101-120 (in Russian)]

[3] Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. *Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. St. Petersburg. Lecture Notes in Computer Science 7979,* Springer. 2013, 80-89.

[4] Legalov A.I., Nepomnyaschy O.V., Matkovsky I.V., Kropacheva M.S. Tail Recursion Transformation in Functional Dataflow Parallel Programs. *Automatic Control and Computer Sciences*, 2013, 47(7), 366–372.

[5] Легалов А.И. Методы сортировки, полученные из анализа максимально параллельной программы. *Распределенные и кластерные вычисления. Избранные материалы Третьей школы-семинара. Институт вычислительного моделирования СО РАН.* Красноярск, 2004, с. 119-134 [Legalov A.I. Sorting methods obtained from the analysis of the most parallel program. *Distributed and cluster computing. Selected Materials of the Third School Seminar. Institute of Computational Modeling SB RAS.* Krasnoyarsk, 2004, p. 119-134 (in Russian)]

[6] Воеводин В.В., Воеводин Вл.В. *Параллельные вычисления*. СПб.: БХВ Петербург, 2002, 608 с. [Voevodin V.V., Voevodin Vl.V. *Parallel computing.* SPb.: BHV Petersburg, 2002, P. 608 (in Russian)]

[7] De Stefani L. The I/O complexity of hybrid algorithms for square matrix multiplication. *30th International Symposium on Algorithms and Computation,* 2019, (149)33, 1-15.

[8] Legalov A.I., Nepomnyaschy O.V., Matkovsky I.V., Kropacheva M.S. Tail Recursion Transformation in Functional Dataflow Parallel Programs. *Automatic Control and Computer Science,*. 2013, 47(7), 366–372.

[9] Parallel Matrix Multiplication. *The JR Programming Language. The International Series in Engineering and Computer Science.* Springer, Boston, MA, 2009, 774.