Higher School of Economics
National Research University
School of Software Engineering

Технические приемы процедурно-параметрического программирования

А.И. Легалов
П.В. Косов

http://www.softcraft.ru/ppp/

Информация

1. Легалов А.И., Косов П.В. Процедурно-параметрическое расширение языка программирования С. Синтаксис и семантика. <http://www.softcraft.ru/ppp/ppc/>

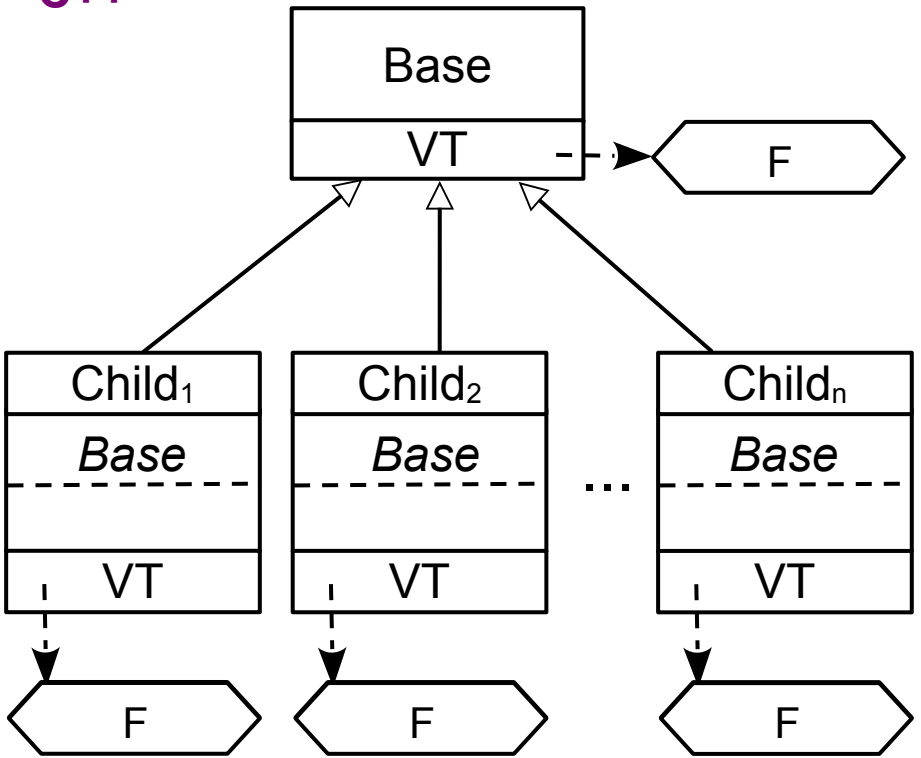
2. Размещение проекта с процедурно–параметрической версией языка программирования С на Гитхаб. <https://github.com/kpdev/llvm-project/tree/pp-extension-v2>

3. Примеры на Гитхаб, написанные с использованием процедурно–параметрической версии языка С. <https://github.com/kreofil/evo-situations>

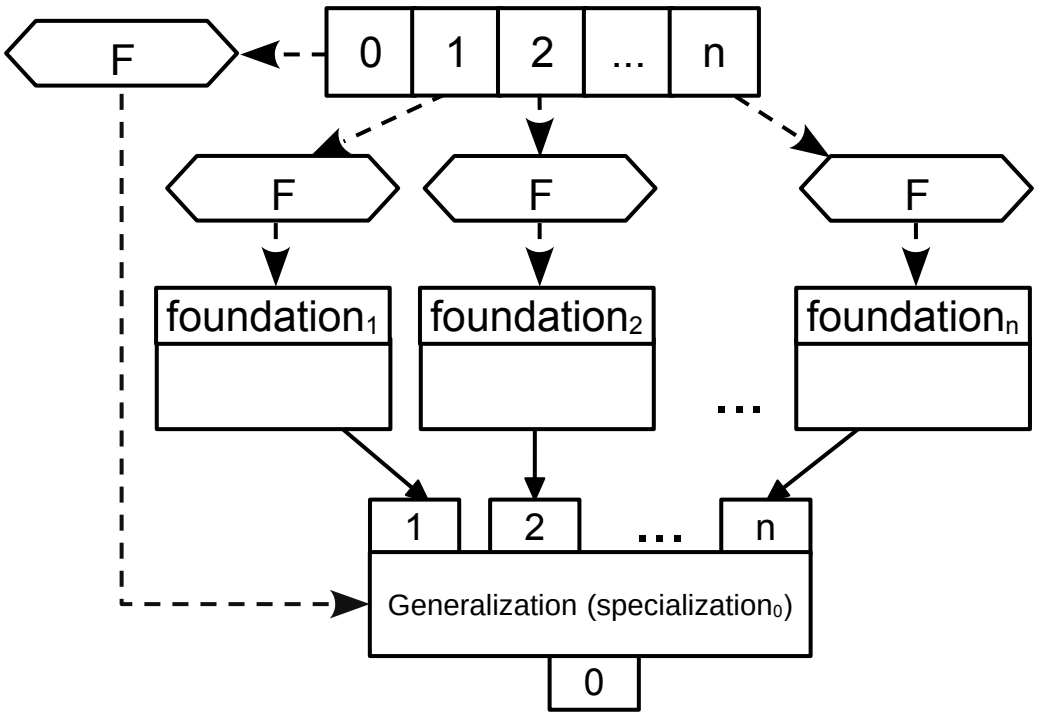
4. Прототип семантической модели (промежуточного представления), разрабатываемый с использованием ППП. <https://github.com/kreofil/ppp-semantic-model>

ООП vs ППП

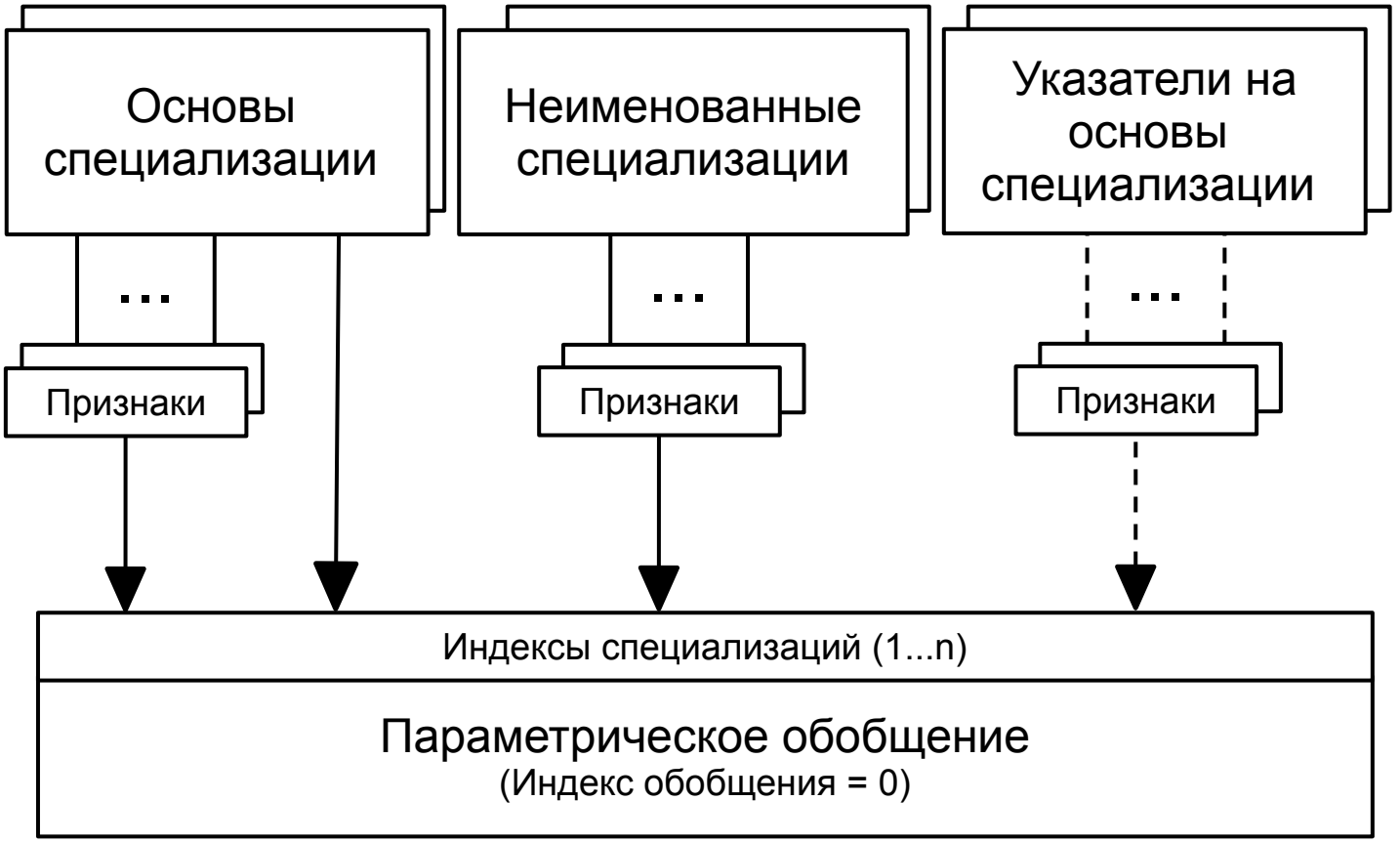
C++



P2C



P2C


Основы специализации
Неименованные специализации
Указатели на основы специализации
...
...
...
Признаки
Признаки
Признаки
Индексы специализаций (1...n)
Параметрическое обобщение
(Индекс обобщения = 0)

ООП vs ППП

C++

P2C

```
class Figure {
public:
  // иденитификация, порождение и ввод фигуры из потока
  static  Figure* In(std::ifstream &ifst);
  // ввод данных из потока
  virtual void InData(std::ifstream &ifst) = 0;
  // вывод данных в стандартный поток
  virtual void Out(std::ofstream &ofst) = 0;
};
```

```
typedef struct Figure {} <> Figure;

// Прототип обобщенной функции ввода фигуры
void FigureIn<Figure *f>(FILE* file);
// Прототип обобщеннай функции вывода фигуры
void FigureOut<Figure *f>(FILE* ofst);

// Обобщающая функция для ввода параметров фигуры
void FigureIn<Figure *f>(FILE* file) = 0;
// Обобщающая функция для вывода параметров фигуры
void FigureOut<Figure *f>(FILE* file) = 0;
```

ООП vs ППП

C++

```
// прямоугольник
class Rectangle: public Figure {
  int x, y; // ширина, высота
public:
  // переопределяем интерфейс класса
  virtual void InData(std::ifstream &ifst);
  virtual void Out(std::ofstream &ofst);
  Rectangle(): x{0}, y{0} {}
};

// треугольник
class Triangle: public Figure {
  int a, b, c; // стороны
public:
  // переопределяем интерфейс класса
  void InData(std::ifstream &ifst);
  void Out(std::ofstream &ofst);
  Triangle(): a{0}, b{0}, c{0} {}
};
```

P2C

```
// прямоугольник
typedef struct Rectangle {
  int x, y; // ширина, высота
} Rectangle;

// треугольник
typedef struct Triangle {
  int a, b, c; // стороны треугольника
} Triangle;

// Фигура - прямоугольник
Figure + < rect: Rectangle; >;

// Фигура - треугольник
Figure + < trian: Triangle; >;
```

ООП vs ППП

P2C

```
void RectangleIn(Rectangle*, FILE*);
void RectangleOut(Rectangle*, FILE*);
// Ввод прямоугольника как фигуры
void FigureIn<Figure.rect *f>(FILE* ifst) {
  RectangleIn(&(f->@), ifst);
}
// Вывод прямоугольника как фигуры
void FigureOut<Figure.rect *f>(FILE* ofst) {
  RectangleOut(&(f->@), ofst);
}

void TriangleIn(Triangle*, FILE*);
void TriangleOut(Triangle*, FILE*);
// Ввод треугольника как фигуры
void FigureIn<Figure.trian *f>(FILE* ifst) {
  TriangleIn(&(f->@), ifst);
}
// Вывод треугольника как фигуры
void FigureOut<Figure.trian *f>(FILE* ofst) {
  TriangleOut(&(f->@), ofst);
}
```

Использование в специализациях обработчиков обобщения

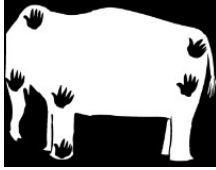
C++

```
// Информация, порождаемая
// родительским методом вывода
void Figure::Out() {
  std::cout
    <<
      "Output of the general figure\n";
}

// Вывод параметров треугольника
void Triangle::Out() {
  Figure::Out();
  std::cout
    << "It is Triangle: a = "
    << a << ", b = " << b
    << ", c = " << c << "\n";
}
```

P2C

```
// Функция обработки обобщения
// независимо от того, какая из
// специализаций поступает на ее вход
void OutGeneralFigure(Figure *f) {
  printf("Output of the general figure\n");
}

// Обобщающая функция для вывода информации
// Через дополнительную функцию - делегата
void Out<Figure *f>() {
  // Запуск обработчика обобщенной фигуры
  OutGeneralFigure(f);
}

//---------------------------------------
// Вывод треугольника-фигуры
void Out<Figure.trian *f>() {
  // Запуск обработчика обобщенной фигуры
  OutGeneralFigure((Figure*)f);
  TriangleOut(&(f->@));
}
```

ППП и диспетчеризация



```
// Обобщающая функция, задающая вход в первую фигуру.
// Вторая фигура передается как обычный параметр
void Multimethod<struct Figure* f1>(Figure* f2, FILE* ofst) = 0;

// Обработчик специализации, когда первая фигура - прямоугольник
void Multimethod<struct Figure.rect* r1>(Figure* f2, FILE* ofst) {
  MultimethodFirstRect<f2>(r1, ofst);
}

// Обработчик специализации, когда первая фигура - треугольник
void Multimethod<struct Figure.trian* t1>(Figure* f2, FILE* ofst) {
  MultimethodFirstTrian<f2>(t1, ofst);
}

// Обобщающая функция, задающая вход во вторую фигуру,
// когда первая фигура уже определена и это прямоугольник
void MultimethodFirstRect<Figure* f2>(Figure.rect* r1, FILE* ofst) = 0;

// Обобщающая функция, задающая вход во вторую фигуру,
// Когда первая фигура уже определена и это треугольник
void MultimethodFirstTrian<Figure* f2>(Figure.trian* t1, FILE* ofst) = 0;
```

https://github.com/kreofil/evo-situations/tree/main/evolution/ppclang-ppp/direct/06-multimethod/06-ppp-mm-tag-disp

ППП и диспетчеризация

```
// Обработчик специализации для двух прямоугольников
void MultimethodFirstRect<Figure.rect* r2>(Figure.rect* r1, FILE* ofst) {
  fprintf(ofst, "Rectangle - Rectangle Combination\n");
}

// Обработчик специализации для прямоугольника и треугольника
void MultimethodFirstRect<Figure.trian* t2>(Figure.rect* r1, FILE* ofst) {
  fprintf(ofst, "Rectangle - Triangle Combination\n");
}

// Обработчик специализации для треугольника и прямоугольника
void MultimethodFirstTrian<Figure.rect* r2>(Figure.trian* t1, FILE* ofst) {
  fprintf(ofst, "Triangle - Rectangle Combination\n");
}

// Обработчик специализации для двух треугольников
void MultimethodFirstTrian<Figure.trian* t2>(Figure.trian* t1, FILE* ofst) {
  fprintf(ofst, "Triangle - Triangle Combination\n");
}
```

ППП и интерфейсы

Стиль С. Использование заголовочных файлов.
Каждый файл на свой комплект прототипов обобщенных функций

```
figure.h
typedef struct Figure {} <> Figure;
inout.h
void In<Figure *x>();  // Обобщающая функция для ввода через интерфейс
void Out<Figure *x>(); // Обобщающая функция для вывода через интерфейс
...
inout-rectangle.h
Figure + < rect: Rectangle*; >; // Привязка прямоугольника через указатель
void In<Figure.rect *x>();  // Ввод прямоугольника через указатель в интерфейсе
void Out<Figure.rect *x>(); // Вывод прямоугольника через InOut
...
inout-rectangle.c
// Ввод прямоугольника через указатель в псевдоинтерфейсе InOut
void In<Figure.rect *x>() {
  RectangleIn(x->@);
}
// Вывод прямоугольника через указатель в псевдоинтерфейсе InOut
void Out<Figure.rect *x>() {
  RectangleOut(x->@);
}
```

https://github.com/kreofil/evo-situations/tree/main/technics/interfaces/c-style

ППП и интерфейсы

Стиль “Go”

```
inout.h
typedef struct InOut {} <> InOut; // структура, обобщающая функции ввода вывода
void In<InOut *x>(); // Обобщающая функция для ввода через интерфейс InOut
void Out<InOut *x>(); // Обобщающая функция для вывода через интерфейс InOut
...
inout-rectangle.h
InOut + < rect: Rectangle*; >; // Привязка прямоугольника через указатель
void In<InOut.rect *x>(); // Ввод прямоугольника через указатель в интерфейсе
void Out<InOut.rect *x>(); // Вывод прямоугольника через InOut
...
inout-rectangle.c
// Ввод прямоугольника через указатель в псевдоинтерфейсе InOut
void In<InOut.rect *x>() {
  RectangleIn(x->@);
}
// Вывод прямоугольника через указатель в псевдоинтерфейсе InOut
void Out<InOut.rect *x>() {
  RectangleOut(x->@);
}
```

https://github.com/kreofil/evo-situations/tree/main/technics/interfaces/go-style

ППП и множественное наследование

Только через подмешивание

```
// Обобщенная константа
typedef struct Constant {}<> Constant;
// Специализация целочисленной константы
Constant + <Int: ConstantInt;>;
...
// Обобщенный тип
typedef struct Type{
  int typeSize;    // Размер типа
  int align;       // Коэффициент выравнивания в памяти (равен степени 2)
}<> Type;
// Специализации типов
Type + <Int: void;>;       // Целочисленный тип
...
// Обобщенная переменная
typedef struct Variable {
  int       memAddr;     // Вид памяти, выделяемой для переменной
  Type      *varType;    // Тип переменной
  Constant  *varValue;   // Значение переменной
}<> Variable;
// Специализации переменной, определяемые используемой памятью
Variable + <Global:   void;>;  // Глобальная память
Variable + <External: void;>;  // Внешняя память (другой модуль)
...
```

https://github.com/kreofil/ppp-semantic-modelhttps://github.com/kreofil/ppp-semantic-model

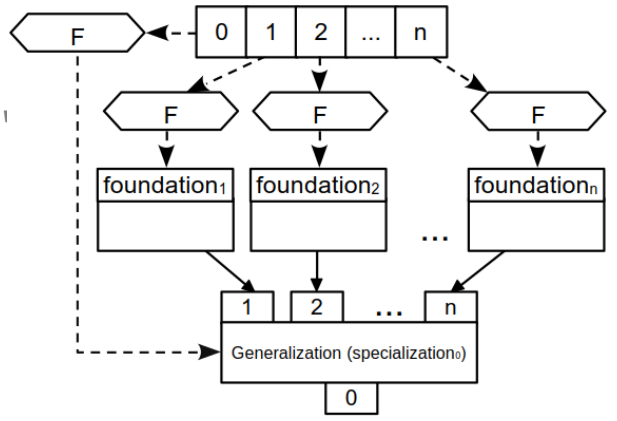
Эволюционное расширение при вводе данных

```
// Ввод параметров одной из фигур из файла
Figure* FigureCreateAndIn(FILE* ifst) {
  Figure *sp;
  int k = 0;
  fscanf(ifst, "%d", &(k));
  switch(k) {
  case 1:
    sp = create_spec(Figure.rect);
    break;
  case 2:
    sp = create_spec(Figure.trian);
    break;
  default:
    return 0;
  }
  FigureIn<sp>(ifst);
  return sp;
}
```

```
1
3 4
2
1 1 1
1
30 40
2
2 2 1
1
13 14
2
10 10 10
1
330 49
2
3 4 5
```

Эволюционное расширение при вводе данных

```
// Ввод параметров одной из фигур из файла
// ЧислоСпециализаций =
//    int get_spec_size "(" ИмяОбобщения ")".
// ПолучениеУказателяНаСпециализацию =
//    УказательНаОбобщение get_spec_ptr
//       "(" ИмяОбобщения,  ИндексСпециализации ")"
Figure* FigureCreateAndIn(FILE* ifst) {
  int figSpecSize = get_spec_size(Figure);
  Figure *sp;
  int k = 0;
  fscanf(ifst, "%d", &(k));
  for(int i = 1; i <= figSpecSize; i++) {
    Figure* pFig = get_spec_ptr(Figure, i);
    sp = FigureCreateUseTag<pFig>(k);
    if(sp != NULL) break;
  }
  if(sp == NULL) {
    printf("Incorrect pointer to input figure\n");
    exit(13);
  }
  FigureIn<sp>(ifst);
  return sp;
}
```



P2C

```
1
3 4
2
1 1 1
1
30 40
2
2 2 1
1
13 14
2
10 10 10
1
330 49
2
3 4 5
```

https://github.com/kreofil/evo-situations/tree/main/evolution/ppclang-ppp/direct/01-start/01-ppp-start-tag-in-fig

Эволюционное расширение при вводе данных

P2C

```
// Обобщающая функция, которая по указателю на специализацию
// и значению из файла создает специализированную фигуру
Figure* FigureCreateUseTag<Figure *pFig>(int k) {
  printf("Unnown figure type for k = %d\n", k);
  exit(13);
}

// Создание прямоугольника как фигуры
Figure* FigureCreateUseTag<Figure.rect *pFig>(int k) {
  if(k == 1) {
    return create_spec(Figure.rect);
  } else {
    return NULL;
  }
}

// Создание треугольника как фигуры
Figure* FigureCreateUseTag<Figure.trian *pFig>(int k) {
  if(k == 2) {
    return create_spec(Figure.trian);
  } else {
    return NULL;
  }
}
```

```
1
3 4
2
1 1 1
1
30 40
2
2 2 1
1
13 14
2
10 10 10
1
330 49
2
3 4 5
```

Сравнение типов специализаций с использованием признаков

```
// Обобщенная функция определяющая, является ли фигура прямоугольником
// ИдентичностьСпециализаций =
//   int spec_index_cmp "(" УказательНаОбобщение,  УказательНаОбобщение ")"
_Bool isFigureRectangle(Figure * f) {
  struct Figure.rect figRect;
  if(spec_index_cmp(&figRect, f) >= 0) {
    return 1;
  }
  return 0;
}
// Вывод прямоугольников из контейнера в указанный поток
void ContainerRectangleOnlyOut(Container *c, FILE* ofst) {
  int rectCount = 0;
  for(int i = 0; i < c->len; i++) {
    if(isFigureRectangle(c->cont[i])) {
      FigureOut<c->cont[i]>(ofst);
      ++rectCount;
    }
  }
  fprintf(ofst, "Container contains %d rectangles.\n", rectCount);
}
```

https://github.com/kreofil/evo-situations/tree/main/evolution/ppclang-ppp/direct/05-rectangles-only-out/05-ppp-rect-only-out-tag-index-cmp

Сопоставление с образцом?

Имитация эволюционно расширяемого перечислимого типа

```
// Основа простого эволюционно расширяемого перечислимого типа
typedef struct Enum{}<> Enum;
// Специализации перечислимого типа
Enum + <one:   void;>;    Enum + <two:   void;>;    Enum + <three: void;>;

// Обобщенная функция вывода значения перечислимого типа
void OutEnum<Enum* e>() {printf("It is Zero. Size = %lu\n", sizeof(Enum));}
// Обработчики специализаций
void OutEnum<Enum.one* e>() { printf("It is One. Size = %lu\n", sizeof(*e)); }
void OutEnum<Enum.two* e>() { printf("It is Two. Size = %lu\n", sizeof(*e)); }
void OutEnum<Enum.three* e>() { printf("It is Three. Size = %lu\n", sizeof(*e));}

int main() {
  struct Enum e0;          OutEnum<&e0>();
  struct Enum.one e1;      OutEnum<&e1>();
  struct Enum.two e2;      OutEnum<&e2>();
  struct Enum.three e3;    OutEnum<&e3>();

  init_spec(Enum.one, &e3);   OutEnum<&e3>();
  init_spec(Enum.two, &e0);   OutEnum<&e0>();
  return 0;
}
```

```
It is Zero. Size = 8
It is One. Size = 8
It is Two. Size = 8
It is Three. Size = 8
It is One. Size = 8
It is Three. Size = 8
It is Two. Size = 8
```

https://github.com/kreofil/evo-situations/tree/main/technics/enums/enum-01

Использование специализаций одинакового размера

```
// структура, обобщающая фигуры
typedef struct Figure {} <> Figure;

// Специализация для формирования фиктивной заглушки
Figure + <empty: void*;>;

// Простейший контейнер
enum {max_len = 100}; // максимальная длина
typedef struct Container {
  int len; // текущая длина
  struct Figure.empty cont[max_len];
} Container;

void ContainerClear(Container *c) { // Очистка контейнера от элементов
  for(int i = 0; i < c->len; i++) {
    struct Figure.empty *pfr = &(c->cont[i]);
    free(pfr->@);
  }
  ContainerInit(c);
}
```

https://github.com/kreofil/evo-situations/tree/main/evolution/ppclang-ppp/direct/01-start/01-ppp-start-init-empty

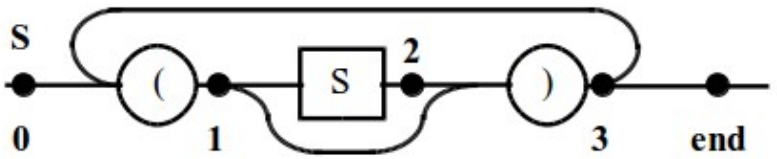
Использование специализаций одинакового размера

```
// Ввод параметров одной из фигур из файла
void FigureCreateAndIn(Figure* f, FILE* ifst) {
  Rectangle *pr;  // Для создания прямоугольника
  Triangle  *pt;  // Для создания треугольника
  int k = 0;
  fscanf(ifst, "%d", &(k));
  switch(k) {
  case 1:
    pr = malloc(sizeof(Rectangle));
    struct Figure.rect* pfr = (struct Figure.rect*)f;
    init_spec(Figure.rect, pfr);
    pfr->@ = pr;
    break;
  case 2:
    pt = malloc(sizeof(Triangle));
    struct Figure.trian* pft = (struct Figure.trian*)f;
    init_spec(Figure.trian, pft);
    pft->@ = pt;
    break;
  default:
    perror("Incorrect figure tag\n");
    exit(13);
  }
  FigureIn<f>(ifst);
}
```

```
ИнициализацияСпециализации =
  void init_spec "(" ИмяОбобщения["." ПризнакСпециализации]
                 "," УказательНаОбластьПамяти ")".
```

```
// Ввод содержимого контейнера
void ContainerIn(Container* c, FILE* ifst)
{
  while(!feof(ifst))  {
    FigureCreateAndIn
     ((Figure*)&(c->cont[c->len]), ifst);
    c->len++;
  }
}
```

Построение иерархически порождаемого конечного автомата

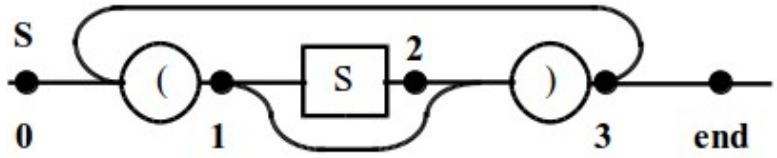


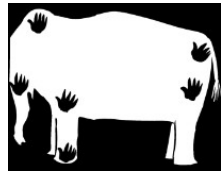

```
// Функция, реализующая распознавание нетерминала S.
_Bool S() {
//_0: // Начало диаграммы
  if(str[i] == '(') { i++; goto _1; }
  return 0; // Первый символ цепочки некорректен
_1: // Точка 1 диаграммы
  if(str[i] == ')') { i++; goto _3; }
  if(S()) { goto _2; }
  erFlag++;
  Printf(
    "Position %d, Error 1: I want closed or next opened bracket!\n", i);
  return 0;
_2: // Точка 2 диаграммы
  if(str[i] == ')') { i++; goto _3; }
  erFlag++;
  printf("Position %d, Error 2: I want closed bracket!\n", i);
  return 0;
_3: // Точка 3 диаграммы
  if(str[i] == '(') { i++; goto _1; }
  goto _end;
_end: // Точка end диаграммы
  return 1;
}
```

http://www.softcraft.ru/translat/lect/t09/#10
https://github.com/kreofil/evo-situations/tree/main/technics/bracket-state-machine/BracRecWirthC

Построение иерархически порождаемого конечного автомата

```
// Обобщение, определяющее состояния автомата S
typedef struct StateS{}
    <S1, S2, S3, True, False: void;> StateS;
//--------------------------------------------------------------------------
// Обобщенная функция реализует начальное состояние
_Bool CalcStateS<StateS* state>() { //_0: // Начало диаграммы
  if(str[i] == '(') { i++; init_spec(StateS.S1, state); return 1; }
  init_spec(StateS.False, state);
  return 0; // Первый символ цепочки некорректен
};
_Bool CalcStateS<StateS.S1* state>() { // _1: // Точка 1 диаграммы
  if(str[i] == ')') { i++; init_spec(StateS.S3, state); return 1; }
  if(S()) { init_spec(StateS.S2, state); return 1; }
  erFlag++;
  printf(
    "Position %d, Error 1: I want closed bracket or next opened bracket!\n", i);
  init_spec(StateS.False, state);
  return 0;
}
_Bool CalcStateS<StateS.S2* state>() { // _2: // Точка 2 диаграммы
  if(str[i] == ')') { i++; init_spec(StateS.S3, state); return 1; }
  erFlag++;
  printf("Position %d, Error 2: I want to close bracket!\n", i);
  init_spec(StateS.False, state);
  return 0;
}
_Bool CalcStateS<StateS.S3* state>() { // _3: // Точка 3 диаграммы
  if(str[i] == '(') { i++; init_spec(StateS.S1, state); return 1; }
  init_spec(StateS.True, state);
  return 0;
}
_Bool CalcStateS<StateS.False* state>() { return 0; }

_Bool CalcStateS<StateS.True* state>() { return 1; }
```



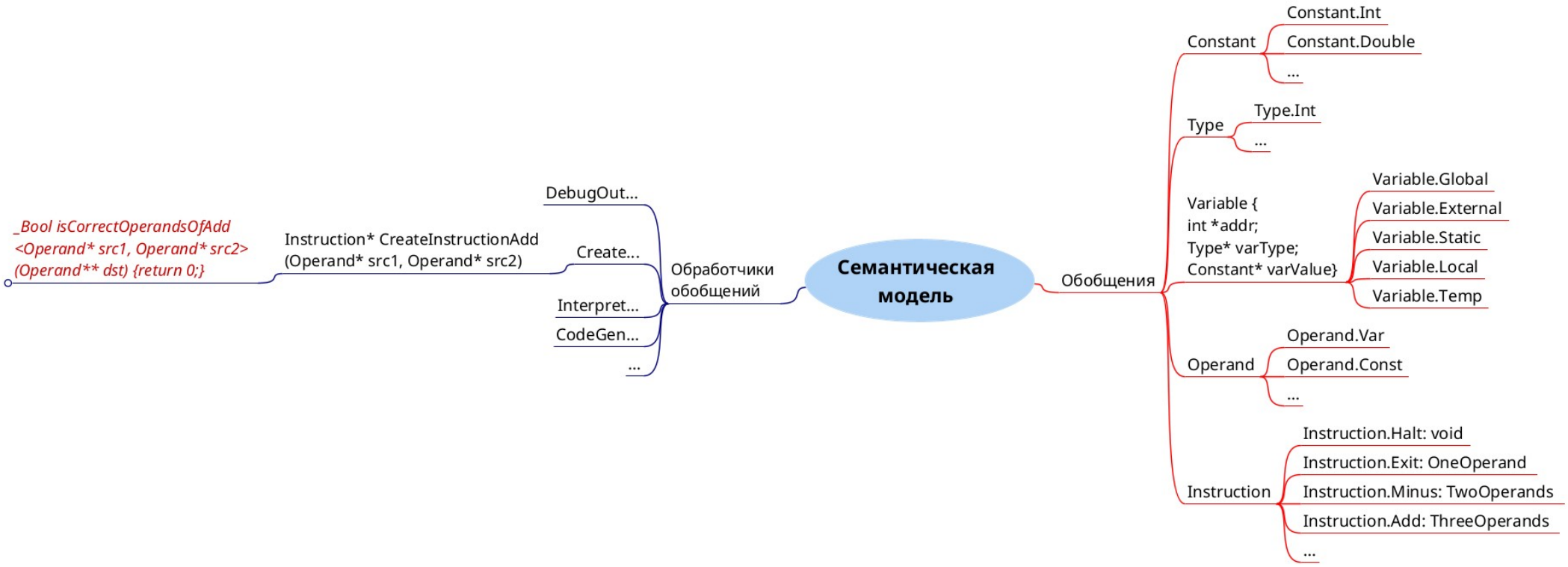


Динамическая декларативность?

```
//-------------------------------------------------
// Автомат, реализующий распознавание нетерминала S.
_Bool S() {
  struct StateS state;
  while(CalcStateS<&state>());
  return CalcStateS<&state>();
}
```

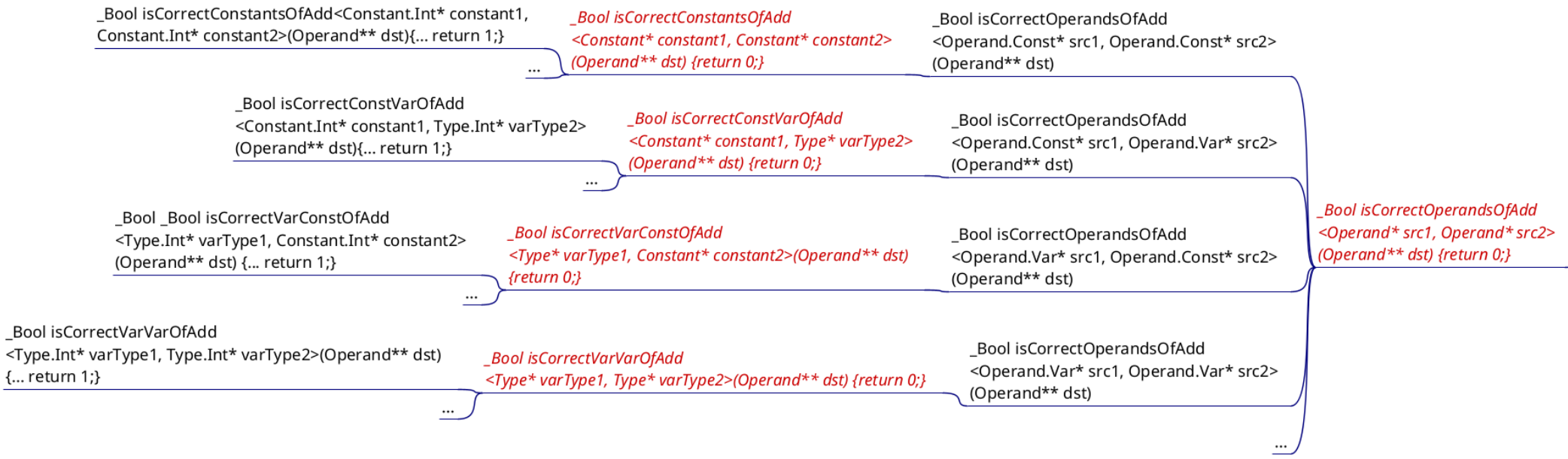
https://github.com/kreofil/evo-situations/tree/main/technics/bracket-state-machine/BracRecWirthPpcBool

Построение семантической модели



https://github.com/kreofil/ppp-semantic-modelhttps://github.com/kreofil/ppp-semantic-model

Построение семантической модели



Имитация ОО полиморфизма через параметрические таблицы

```
// Класс, обобщающает все имеющиеся фигуры.
// Является абстрактным, обеспечивая, тем самым, проверку интерфейса
class Figure {
public:
  // Идентификация, порождение и ввод любой фигуры из потока
  static  Figure* In(std::ifstream &ifst);
  // Метод ввода данных созданной фигуры через параметрическую таблицу (полиморфизм)
  void InFP(std::ifstream &ifst);
  // Метод вывода фигуры через параметрическую таблицу
  void OutFP(std::ofstream &ofst);
  // Формирование параметрических отношений вместо виртуальных методов
  int specTag; // Признак специализации, доступный из обобщения
  // Счетчик числа зарегистрированных специализаций
  static int specCounter;
};

// Общее описание типа обобщающих функций
typedef void (*InDataP)(Figure* f, std::ifstream &ifst);
typedef void (*OutDataP)(Figure* f, std::ofstream &ofst);

// Параметрическая таблица для функций ввода
extern std::vector<InDataP> InP;
// Параметрическая таблица для функций вывода
extern std::vector<OutDataP> OutP;
```

https://github.com/kreofil/evo-situations/tree/main/evolution/cpp-opp-model/direct/01-start/01-opp-direct-inhierence

Имитация ОО полиморфизма через параметрические таблицы

```
// Счетчик числа зарегистрированных специализаций
int Figure::specCounter = 0;

//-------------------------------------------------------------------------
// Параметрическая таблица для функций ввода
std::vector<InDataP> InP;
// Параметрическая таблица для функций вывода
std::vector<OutDataP> OutP;

//-------------------------------------------------------------------------
// Метод ввода фигуры через параметрическую таблицу
void Figure::InFP(std::ifstream &ifst) {
  InP[specTag](this, ifst);
}

//-------------------------------------------------------------------------
// Метод вывода фигуры через параметрическую таблицу
void Figure::OutFP(std::ofstream &ofst) {
  OutP[specTag](this, ofst);
}
```

Имитация ОО полиморфизма через параметрические таблицы

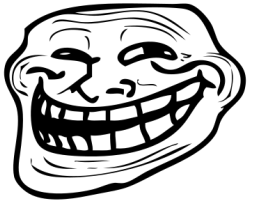
```
//-----------------------------------------------------------
// прямоугольник
class Rectangle: public Figure {
  int x, y; // ширина, высота
public:
  Rectangle();

  // Функция ввода содержимого прямоугольника,
  // подключаемая через параметрическую таблицу
  static void InRP(Figure* f, std::ifstream &ifst);
  // Функция вывода содержимого прямоугольника,
  // подключаемая через параметрическую таблицу
  static void OutRP(Figure* f, std::ofstream &ofst);

  // Признак фигуры
  static int tag;
};
```

Имитация ОО полиморфизма через параметрические таблицы

```
int Rectangle::tag = -1; // Значение признака специализации до регистрации

// Конструктор, обеспечивающий формирование прямоугольника,
// установку признака и регистрацию в параметрических таблицах
Rectangle::Rectangle(): x{0}, y{0} {
  if(tag == -1) {
    // Установка тега и размещение в параметрических таблицах
    tag = specCounter++;
    InP.push_back(Rectangle::InRP);
    OutP.push_back(Rectangle::OutRP);
  }
  specTag = tag;  // Инициализация собственного тега объекта
}

// Ввод содержимого прямоугольника, подключаемый через параметрическую таблицу
void Rectangle::InRP(Figure* f, std::ifstream &ifst) {
  Rectangle* r = reinterpret_cast<Rectangle*>(f);
  ifst >> r->x >> r->y;
}

// Вывод содержимого прямоугольника, подключаемый через параметрическую таблицу
void Rectangle::OutRP(Figure* f, std::ofstream &ofst) {
  Rectangle* r = reinterpret_cast<Rectangle*>(f);
  ofst << "It is Rectangle: x = " << r->x << ", y = " << r->y << "\n";
}
```

Кастрация ППП?

ППП и процесс разработки



ООП:

1) Прецеденты – это функции. Зачем их отображать в классы? Искусственная привязка к конструкциям, ограничивающим взаимодействие. Классы не всегда эффективно отображают сложные отношения.
2) Прямое отображение не приветствуется, так как не позволяет достичь требуемых критериев качества. Приходится трансформировать в абстракции, не связанные с предметной областью. Паттерны.

ППП:

1) Функциональность предметной области представлена напрямую.
2) Многие отображения в код реализуются напрямую или более гибко.
3) Что мешает использовать ПП язык вместо ОО языка в ОО проектах? (только отсутствие нормального языка?)
4) Возможное дополнительное влияние на процесс проектирования?

Благодарю за внимание!

http://www.softcraft.ru/ppp/