УДК 004.4'42, 004.43

ПРОЦЕДУРНО–ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ В ЯЗЫКЕ С ДЛЯ ПОВЫШЕНИЯ НАДЕЖНОСТИ ПРОГРАММ

А.И. Легалов

Национальный исследовательский университет «Высшая школа экономики»

Россия, 101000, Москва, ул. Мясницкая, 20

E-mail: alegalov@hse.ru

П.В. Косов

Национальный исследовательский университет «Высшая школа экономики»

Россия, 101000, Москва, ул. Мясницкая, 20

E-mail: kosov_@mail.ru

Ключевые слова: язык программирования, компиляция, процедурно-параметрическое программирование, полиморфизм, эволюционная разработка программного обеспечения, надежность программного обеспечения.

Аннотация: Разработка программ зачастую связана с инкрементальным расширением функциональности. Для повышения надежности в этом случае необходимо минимизировать изменение ранее написанного кода. Для инструментальной поддержки эволюционной разработки предложена процедурно-параметрическая парадигма, расширяющая возможности процедурного подхода. Она обеспечивает безболезненное расширение как данных, так и функций, используя при этом статическую типизацию вместо разыменования типов. В работе рассматривается включение процедурно-параметрических механизмов в язык С, который часто используется для параллельного и распределенного программирования. Предлагаются дополнительные синтаксические конструкции, ориентированные на поддержку подхода. К ним относятся: параметрические обобщения, специализации обобщений, обобщающие функции, обработчики специализаций. Рассматриваются их возможности. Описаны ситуации, при которых возможно повышение надежности программы за счет использования процедурно-параметрического полиморфизма.

1. Введение

При разработке программных систем необходимо учитывать различные критерии качества. Расширение программы без изменения ранее написанного кода является одним из них. Оно обуславливается как неполным знанием окончательной функциональности программ во время ее создания и начальной эксплуатации, так и необходимостью ускорить выход продукта на рынок за счет реализации только базового набора функций с последующим его наращиванием. В подобных

ситуациях добавление новых программных конструкций в код, сопровождаемое его изменением, зачастую ведет к появлению ошибок и непредсказуемому поведению.

Расширение программ во многом связано с использованием динамического связывания программных объектов, обеспечивающего обработку альтернативных ветвей программы во время ее выполнения. При этом могут использоваться как явный анализ альтернатив, так и неявный. Последний определяет динамический полиморфизм. Он отличается от статического полиморфизма, при котором альтернативные вычисления выявляются во время компиляции.

Изначально реализация динамического полиморфизма, была предложена для объектно–ориентированной (ОО) парадигмы. В ОО языках со статической типизацией ключевым решением стало совместное использование механизмов наследования и виртуализации. Наследование обеспечило идентичность интерфейсов родительского и дочернего классов, а виртуализация позволила подменять методы в дочерних классах. В качестве примера можно привести языки C++ [1], Java [2] и другие.

Поддержку динамического полиморфизма, основанную на иных принципах, стали включать и в процедурные языки. В языке Go [3], реализован механизм интерфейсов, позволяющий использовать в качестве обработчиков альтернатив функции, связанные со структурами данных. Аналогичный механизм на основе типажей реализован в языке программирования Rust [4]. Вместе с тем представленные выше подходы напрямую не поддерживают эволюционного расширения программ в случае множественного полиморфизма, связанного с реализацией мультиметодов. Достижение необходимого эффекта возможно только написанием дополнительного кода.

Для инструментальной поддержки эволюционно расширяемого множественного полиморфизма была предложена процедурно-параметрическая (ПП) парадигма программирования [5], что позволило расширить возможности процедурного подхода. Решение базируется на параметрическом механизме формирования отношений между данными и обрабатывающими их процедурами, который может быть реализован различными способами [6]. Это обеспечивает безболезненное расширение как данных, так функций в статически типизированных языках программирования. Для апробации идеи разработан язык O2M, расширяющий язык программирования Оберон-2 [7]. Проведенные на его основе эксперименты позволили проанализировать возможности процедурно-параметрического программирования (ППП). Было показано, что подход может быть достаточно безболезненно интегрирован как в уже существующие языки процедурного и функционального программирования, так и использоваться при разработке новых языков. Показано, что ПП парадигма обеспечивает более гибкое эволюционное расширение программ по сравнению с другими методами поддержки динамического полиморфизма [8].

В настоящее время ведется работа, связанная с включением механизмов ППП в язык программирования C, который входит в пятерку наиболее популярных языков по различным рейтингам. Язык обладает гибкостью и относительной простотой, широко используясь в предметных областях, где ОО подход не является эффективным. К таким областям относятся параллельное и распределенное программирование. В частности большинство операционных систем в настоящее время написаны на C. Вместе с тем, многие решения, применяемые при разработке

программ на языке C, могут приводить к ошибкам за счет отсутствия контроля ряда ситуаций во время компиляции. В частности, можно отметить следующее:

- добавление новый альтернатив в уже написанный код ведет к модификации объединений этих альтернатив, их признаков, а также функций их обработки, что может привести к некорректным модификациям уже написанного кода;
- признаки альтернатив, являющиеся по сути их ключами, непосредственно не связаны с самими альтернативами, что является источником потенциальных ошибок, когда обработчик конкретной альтернативы может быть случайно привязан к другом признаку;
- при подключении альтернатив к общему указателю часто осуществляется разыменование типов (использование указателя `void*`), что очень часто является источником ошибок;
- разыменование типов также часто используется при передаче альтернатив в функции, что требует восстановление типов внутри этих функций и не всегда осуществляется корректно.

Многие из этих проблем эффективно решаются в ОО языках программирования за счет использования полиморфизма, а также в языках Go и Rust. ПП парадигма наряду с решением этих проблем позволяет также эффективно и эволюционно поддерживать расширение мультиметодов.

Основные понятия процедурно-параметрической парадигмы, а также синтаксис и семантика ПП механизмов, интегрируемых с языком программирования C представлены в работе [9]. Ниже на примерах показано, каким образом использование ППП позволяет повысить надежность программ в представленных выше ситуациях.

2. Добавление новых альтернатив вместо использования объединений

Существуют различные подходы к формированию данных, описывающих множество альтернатив, которые также называются обобщениями. Альтернативные конструкции, включенные в обобщения являются специализациями. Часто основами специализации являются независимые данные описывающие независимые понятия. В качестве примеров основ специализаций, можно привести представления треугольника, прямоугольника и круга:

```
typedef struct Rectangle{int x, y;} Rectangle;  // Прямоугольник
typedef struct Triangle{int a, b, c;} Triangle; // Треугольник
```

Одним из вариантов обобщения может являться структура, объединяющая эти основы и дополнительно содержащая признак фигуры, задаваемый перечислением:

```
// Значения ключей для каждой из фигур
typedef enum Key {RECTANGLE, TRIANGLE} Key;
typedef struct Figure { // Обобщенная фигура
    Key k; // ключ
    union { // Используемые альтернативы
```

```
            Rectangle r;
            Triangle t;
        };
    } shape;
```

Подключение новых специализаций, например, круга, связано с необходимостью изменять как ключи, задающие признаки фигуры, так и само обобщение, в которую включается новая специализация.

```
    typedef struct Circle{int r;} Circle;           // Круг
    // Значения ключей для каждой из фигур
    typedef enum key {RECTANGLE, TRIANGLE, CIRCLE} key;
    typedef struct Figure { // Обобщенная фигура
    key k; // ключ
    union { // Используемые альтернативы
        Rectangle r;
        Triangle t;
        Circle c;
    };
} shape;
```

Формирование ПП обобщения может осуществляться различными способами [9]. В качестве примера можно рассмотреть использование имени типа в качестве признака. При этом изначально обобщение может быть создано без подключения специализаций (хотя в общем случае может содержать их):

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

Конструкция в виде пустых угловых скобок указывает на то, что в дальнейшем обобщение может быть расширено путем добавления в эти скобки новых специализаций. Специализации могут, опираясь на описание обобщения, добавляться в различных единицах компиляции в ходе эволюционного расширения программы. Таким образом в обобщение могут быть добавлены независимо прямоугольник, треугольник и круг:

```
    // Прямоугольник и треугольник как специализации
    Figure + <Rectangle; Triangle;>;
    Figure + <Circle;>;     // Круг как специализация
```

Формируемые при этом программные объекты, в отличии от производных классов при ОО подходе, имеют единый тип `Figure`, отличаясь только типом специализации, которая непосредственно связана с используемой при этом основой. Это обеспечивает однозначность между признаком специализации и ее содержимым в отличие от использования объединения.

3. Использование указателей на обобщения вместо разыменования

Часто для эволюционного расширения программы на языке С используется разыменование. Например, добавление новых специализаций можно реализовать путем их формирования, независимо от обобщения, или вообще игнорировать

обобщение как специально сформированную сущность. Для этого часто используется эквивалентность полей структуры в памяти. Можно в каждой из специализаций поставить на первое место поле ее признака с последующим размещением необходимых данных:

```
typedef struct FigRectangle {Key k; Rectangle r;} FigRectangle;
typedef struct FigTriangle {Key k; Triangle t;} FigTriangle;
typedef struct FigCircle {Key k; Circle c;} FigCircle;
```

Подключение любой переменной одного из указанных типов может осуществляться через указатель на `void*` с последующей проверкой ключа за счет приведения к любому из типов, а затем возможно повторному приведению к выявленному типу с обработкой конкретной фигуры. Это снижает надежность программы.

ПП решение позволяет сформировать указатель на обобщенную фигуру с последующим выявлением специализации как явной проверкой ее типа, так и с использованием ПП полиморфизма. В этом случае любая сформированная специализированная переменная может быть подключена через указатель на фигуру. Например:

```
// Специализированные треугольники t1 и t2:
Figure<Triangle> t1 = {}<{3,4,5}>, t2;
// Круг c1, указатель на него c2, массив кругов c
Figure<struct Circle> c1 = {}<{10}>, *pc1 = &c1, c[10];
// Прямоугольники r1-r3 и указатель pr1 на один из них
Figure<Rectangle> r1, r2[3] = {{}<{1,2}>,{}<{3,4}>,{}<{5,6}>},
                  r3 = <{7,4}>, *pr1 = &r1;
// Указатель pf1 на t1 и указатель на указатель ppf1
struct Figure *pf1 = &t1, **ppf1 = &pf1;
```

Корректность подключения к данным указателям проверяется во время компиляции программы, что повышает надежность по сравнению с традиционными решениями.

Явно выяснить тип специализации, подключенной к обобщенному указателю можно с использованием специально введенной функции:

```
_is_spec("Указатель на обобщение", "Признак специализации")
```

При необходимости можно в дальнейшем осуществить явное приведение типа. Однако для организации надежных вычислений лучше использовать обобщенные функции и обработчики специализаций [9].

4. Замена централизованного обработчика альтернатив на полиморфную обработку

Обработка альтернатив в языке С обычно связана с использованием явного и централизованного перебора признаков в условных операторах или переключателях с последующим выполнением операций, присущих этой альтернативе. При этом отсутствие непосредственной зависимости между признаком и ее обработчиком может привести к тому, что соответствующему признаку может ошибочно сопоставлен не тот обработчик специализации. Добавление новой альтернативы ведет к непосредственном изменению кода внутри обработчика обобщения, что также связано с риском неправильного сопоставления новых специализаций с

признаками. Подобные ошибки обычно выявляются только на этапе выполнения программы.

ПП парадигма предлагает использование обобщающих параметрических функций, поддерживающих ПП полиморфизм. Их сигнатуры определяют единый интерфейс для обработчиков параметрических специализаций. Каждая обобщающая функция имеет от одного до нескольких формальных параметров, задаваемых в виде указателей на обобщения. При вызове такой функции в нее передаются ссылки на специализированные переменные, комбинации которые и определяют конкретный обработчик специализации. Например, обобщающая функция вывода на печать геометрической фигуры может быть представлена следующим образом:

```
void PrintFigure<Figure *f>() = 0;
```

Обобщенные параметры задаются в угловых скобках (прочие параметры, при их наличии, как обычно располагаются в круглых скобках). Отсутствие тела в данном случае говорит о том, что необходимо написать обработчики специализаций для всех конкретных фигур. Вместо этого может быть задан обработчик по умолчанию, который вызывается в тех случаях, когда для подставляемой комбинации специализаций не будет реализован свой обработчик. Простейшим вариантов подобного обработчика в таком случае может служить вызов функции прерывания программы:

```
void PrintFigure<Figure *f>() {
    printf("Incorrect Argument!!!\n")
    exit(-1);
}
```

Выбор обработчика специализации осуществляется в зависимости от значений специализированных аргументов. Они могут располагаться в разных единицах компиляции, добавляясь, например, по мере создания очередной специализации. Для обобщающей функции печати фигуры обработчики специализаций выглядят следующим образом:

```
// Вывод параметров прямоугольника
void PrintFigure<struct Rectangle *r>() {
    printf("Rectangle: x = %d, y = %d", r->@x, r->@y);
}
// Вывод параметров треугольника
void PrintFigure<struct Triangle *t>() {
    printf("Triangle: a = %d, b = %d, c = %d",
           (*t).@a, (*t).@b, (*t).@c);
}
// Вывод параметров круга
void PrintFigure<struct Circle *c>() {
    printf("Circle: r = %d", c->@r);
}
```

Следует отметить, что символ `@`, определяющий доступ к полям специализаций, ставится после точки или знака указателя.

Вызов обобщенной функции по сути запускает обработчик для конкретной комбинации специализаций. Он отличается от вызова обычной функции только наличием списка дополнительных фактических параметров, указывающих на специализации, по которым автоматически осуществляется выбор обработчика, что и

определяет динамический полиморфизм. В качестве примеров можно привести вызов функции печати фигуры для различных выше описанных специализированных переменных:

```
PrintFigure<&t1>(); // печать треугольника t1
PrintFigure<&c1>(); // печать круга c1
PrintFigure<pc1>(); // печать круга c1 через указатель pc1
PrintFigure<&c[5]>(); // печать круга c[5]
PrintFigure<c+5>(); // печать круга c[5] через смещение указателя
// печать треугольника t1 через обобщенный указатель pf1
PrintFigure<pf1>();
// печать треугольника t1 через указатель на указатель *ppf1
PrintFigure<*ppf1>();
```

5. Заключение

Предлагаемое расширение языка программирования C конструкциями, обеспечивающими инструментальную поддержку процедурно–параметрической парадигмы программирования, позволяет разрабатывать более надежные и эволюционно расширяемые программы. При этом обеспечивается добавление новых альтернативных данных, а также функций, позволяющих безболезненно для ранее написанного кода использовать множественный полиморфизм. Проведенное моделирование вносимых изменений, реализованное на языке программирования C в рамках операционной системы Linux подтверждает возможности такой реализации [9].

Реализация расширения осуществляется с использованием семейства компиляторов clang [10], что позволяет изменять как синтаксический анализатор, так и абстрактное синтаксическое дерево, адаптируя их под новые конструкции.

Список литературы

1. Gregoire M. Professional C++. John Wiley & Sons, 2018. 1122 p.
2. SciorеE. Java Program Design. Apress Media, 2019. 1122 p.
3. FreemanA. Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang. Apress, 2022. 1105 p.
4. Blandy J., Orendorff J., Tindall L.F. Programming Rust. O'Reilly Media, 2021. 735 p.
5. Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. 43 с.
6. Легалов И.А. Применение обобщенных записей в процедурно–параметрическом языке программирования // Научный вестник НГТУ. 2007. № 3 (28). С. 25–38.
7. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования // Научный вестник НГТУ. 2003. № 2 (15). С. 25–38.
8. Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно-параметрического подхода // Вычислительные технологии. 2016. Т. 21, № 3. С. 56–69.
9. Легалов А.И., Косов П.В. Расширение языка C для поддержки процедурно-параметрического полиморфизма. Моделирование и анализ информационных систем. 2023. Vol. 30, No. 1. P. 40–62.
10. Clang: A c language family frontend for llvm. `https://clang.llvm.org/`.