УДК 681.3.07

Применение обобщенных записей в процедурно-параметрическом языке программирования

И.А. ЛЕГАЛОВ

Рассматривается применение обобщенных записей вместо процедурно-параметрических обобщений в рамках процедурно-параметрической парадигмы программирования. Обобщенная запись использует как основу обычную запись, расширяясь на конце процедурно-параметрическим обобщением. Это повышает эффективность создания новых типов данных и обеспечивает более гибкое построение эволюционно расширяемых программ.

1. ВВЕДЕНИЕ

Эволюционная разработка в настоящее является неотъемлемой чертой многих программных проектов. Это обусловлено как неполным знанием структуры будущей программы во время проектирования и начальной эксплуатации, так необходимостью ускорить ее выход на рынок. Эволюционное программирование хорошо согласуется с инкрементальным проектированием, так как позволяет добавлять новый код без изменения написанного.

Наличие инструментальной поддержки эволюционной разработки во многом способствовало популярности объектно-ориентированного программирования (ООП). В рамках ООП были предложены паттерны [1], обеспечивающие гибкое наращивание программы, за счет динамического полиморфизма, поддерживаемого в современных объектно-ориентированных (ОО) языках сочетанием виртуализации и наследования. Вместе с тем следует отметить, что безболезненное расширение кода в рамках ООП имеет определенные ограничения. В частности, невозможно прямое расширение мультиметодов. Предлагаемые для этого решения [2-4] в основном ведут к введению дополнительных конструктивов и алгоритмов, в которых, чаще всего, объектно-ориентированный стиль смешивается с процедурным.

Во многом эти проблемы ОО подхода обусловлены тем, что мультиметод в большей степени является внешней процедурой, поддерживающей множественный динамический полиморфизм, чем методом, размещаемым внутри класса, обеспечивающего только одиночный динамический полиморфизм. Не спасает положение и использование параметрического программирования, реализуемого, например, в языке C++ [5], посредством шаблонов. Шаблоны поддерживает только статический полиморфизм, который должен быть полностью определен во время компиляции, и не обеспечивают динамического связывания данных в ходе выполнения программы. Следует отметить, что множественный динамический полиморфизм был реализован в языке программирования CLOS [6]. Однако эта реализация оказалась излишне громоздкой, что не привело к ее заимствованию в других языках, в частности, в C++ [7].

Для повышения эффективности инструментальной поддержки множественного динамического полиморфизма была предложена процедурно-параметрическая парадигма [8], расширяющая возможности процедурного подхода в области эволюционного программирования. Ее использование базируется на параметрическом механизме формирования отношений между данными и обрабатывающими их процедурами, который может быть реализован различными способами [9]. Это обеспечивает эволюционную разработку как данных, так процедур и поддерживает механизм строгой типизации, присущий процедурным языкам. Для апробации парадигмы, на базе языка программирования Оберон-2, был разработан язык O2M [10], использование которого позволило

продемонстрировать возможности процедурно-параметрического программирования (ППП) [11].

Спецификой ППП является возможность гибкого изменения состава параметрических обобщений – типов данных, объединяющих в единую категорию множество альтернативных понятий. Обобщения можно создавать как на основе уже существующих понятий, так и безболезненно расширять их за счет включения новых понятий в этой же или других единицах компиляции. Включаемые понятия отличаются признаками и образуют специализации, а существующая инструментальная поддержка обеспечивает ранжирование этих специализаций внутри обобщения, сопоставляя с каждым из них локальный индекс.

Обобщенные параметрические процедуры используют параметрические обобщения в качестве специальных формальных параметров. Множественный полиморфизм обеспечивается за счет использования в качестве параметров нескольких обобщений. При вызове такой процедуры в качестве фактических параметров осуществляется подстановка конкретной комбинации специализаций, признаки которых однозначно определяют алгоритм их обработки. Это позволяет сформировать для каждой из комбинаций специализаций отдельную процедуру – обработчик специализаций. Механизм инструментальной поддержки ППП обеспечивает независимое написание обработчиков специализаций, их добавление при расширении обобщений, контролирует наличие всех требуемых обработчиков. Выполнение указанных функций осуществляется во время компиляции модулей программы и их компоновки. Возможно также окончательное формирование параметрических отношений и во время выполнения. Добавление новых специализаций и их обработчиков может осуществляться в отдельных единицах компиляции, обеспечивая тем самым гибкое эволюционное расширение программы как по данным, так и процедурам.

Вместе с тем, использование «чистых» параметрических обобщений, рассмотренных в работах [8-11] не вполне удобно для программирования, так как в ряде случаев требует использования дополнительных промежуточных структур данных. Для повышения эффективности процедурно-параметрического программирования в работе предлагаются обобщенные записи, в состав которых включается параметрическое обобщение. Рассматриваются особенности реализации и использования обобщенных записей.

2. СПЕЦИФИКА ОБОБЩЕННЫХ ЗАПИСЕЙ

Процедурно-параметрическая парадигма базируется на концепции обобщения как основы для группировки альтернативных специализаций:

ТипОбобщение = CASE [TYPE] [OF [LOCAL] [СписокСпециализаций]]
[ELSE Специализация] END .
СписокСпециализаций =
((СписокПризнаков ":" (Тип | NIL)) | Тип)
{ "|" ((СписокПризнаков ":" (Тип | NIL)) | Тип) } .
СписокПризнаков = идент ["," идент] .
Специализация = (идент ":" (Тип | NIL)) | Тип.

В представленном синтаксическом описании отображаются различные формы задания базовой структуры обобщения, подробно комментируемые в [10-11]. Каждая специализация обобщения отличается от других признаком, автоматически формируемым при добавлении новой специализации. Специализации строятся на основе любых уже существующих типах данных. В качестве примера можно рассмотреть вариант с явным заданием признаков обобщения:

```
(* Прямоугольник со сторонами x и y *)
Rectangle = RECORD x, y: INTEGER END
(* Треугольник со сторонами a, b и c *)
Triangle = RECORD a, b, c: INTEGER END
```

```
(* Обобщение фигуры *)
Figure = CASE OF trian: Triangle | rect: Rectangle END
```

Использование обобщенных процедур позволяет задать обработку фигуры. Например, процедура вывода фигуры может быть представлена следующим образом:

```
PROCEDURE Out {VAR s: Figure}:= 0
```

При этом обобщенные параметры задаются в фигурных скобках (в отличие от прочих параметров, которые, как обычно, при наличии располагаются в круглых скобках). Отсутствие тела в обобщенной процедуре говорит о том, что необходимо написать обработчики специализаций для всех конкретных фигур, представленных параметрическим обобщением. Они могут располагаться в разных единицах компиляции и выглядеть следующим образом:

```
(* Вывод параметров прямоугольника *)
PROCEDURE Out {VAR r(rect): Figure};
BEGIN
  Out.String("Rectangle: x = "); Out.Int(r.x, 0);
  Out.String(", y = "); Out.Int(r.y, 0);
  Out.Ln;
END Out;

(* Вывод параметров треугольника *)
PROCEDURE Out {VAR t(trian): Figure};
BEGIN
  Out.String("Triangle: a = "); Out.Int(t.a, 0);
  Out.String(", b = "); Out.Int(t.b, 0);
  Out.String(", c = "); Out.Int(t.c, 0);
  Out.Ln;
END Out;
```

Обобщение можно изменить добавлением новых специализаций в модулях, импортирующих исходное объявление. Расширение можно использовать и в том модуле, где объявлено обобщение. Его синтаксис имеет следующий вид:

Расширение = Обобщение "+=" СписокСпециализаций.

Использование можно продемонстрировать примером:

```
(* Круг радиуса r *)
Circle = RECORD r: INTEGER END

(* Расширение обобщения *)
Figure += circ: Circle
```

Дополнительная специализация требует добавления нового обработчика, который для процедуры вывода данных будет выглядеть следующим образом:

```
 (* Вывод параметров круга *)
PROCEDURE Out {VAR c(circ): Figure};
BEGIN
  Out.String("Circle: r = "); Out.Int(c.r, 0);
  Out.Ln;
END Out;
```

Аналогичным образом построение обобщенных процедур, а также добавление обработчиков специализаций осуществляется и для мультиметодов [11], что придает процедурно-параметрической парадигме особую гибкость при разработке эволюционно расширяемых программ.

Использование обобщения в качестве связующего программного объекта, сужает возможности его применения. Зачастую в ходе проектирования одновременно приходится не только вводить новые альтернативы, но и дополнительные поля данных, общие для всех существующих альтернатив. Применение обобщения в связке с обычной записью в этом случае выглядит

несколько громоздко. Предположим, что на основе существующего обобщения, требуется создать запись, добавляющую к каждой из фигур целочисленное поле, информирующее, например, о цвете. Для подключения любой из специализаций в обычную запись, несущую информацию о цвете фигуры приходится добавлять поле, указывающее на обобщение, так как прямое включение обобщения в запись невозможно:

```
(* Указатель на обобщенную фигуру *)
PFigure = POINTER TO Figure
(* Первая фигура с цветом *)
ColoredFigure1 = RECORD color: INTEGER; figure: PFigure END;
```

При создании конкретной фигуры в данном случае приходится формировать не только объект типа **ColoredFigure1**, но и специализацию, на которую указывает указатель типа **PFigure**. Помимо этого, приходится связывать экземпляры данных этих двух типов во время выполнения программы.

В связи с этим, для расширения возможностей процедурно-параметрического программирования предлагается ввести обобщенные записи, которые обеспечивают более гибкое использование параметрических обобщений за счет включения структуры последних в обычную запись. Это ведет к модификации синтаксиса последней следующим образом:

ТипОбобщеннаяЗапись = RECORD [СписокПолей {";" СписокПолей}]
(Обобщение | [CASE ИмяОбобщения] END).

Таким образом, параметрическое обобщение располагается в конце записи и позволяет использовать ее поля во всех специализациях, добавляемых в ходе расширения. Допустимость внутри записи только одного обобщения позволяет применять признак специализации для характеристики всей записи. В примере

```
T0 = RECORD x: INTEGER; CASE END;
T0 += y: REAL;
```

T0 – обобщенная запись, которая содержит пустое обобщение. В следующей строке к записи добавляется специализация ***y*** вещественного типа. В результате возможно существование двух альтернативных объектов: записи, состоящей только из поля ***x***, и записи, которая помимо поля ***x*** содержит вещественное поле с признаком ***y***. Обращение к этому полю состоит из идентификатора переменной и идентификатора признака, задаваемого в круглых скобках. Например:

```
VAR v(y): T0
   ...
v(y) = 3.14;
```

Последняя запись является избыточной, так как признак специализированной переменной, зафиксирован во время компиляции и является неизменным. Поэтому, можно использовать и альтернативное обозначение:

```
v() = 3.14;
```

Наряду с непосредственным использованием обобщения в конце записи допускается создавать записи с уже имеющимися обобщениями. В качестве примера можно рассмотреть тип, включающий обобщение геометрической фигуры и добавляющий к каждой из геометрических фигур целочисленное поле, информирующее, например, о цвете фигуры:

```
(* Вторая фигура с цветом *)
ColoredFigure2 = RECORD color: INTEGER; Figure END;
```

Этот подход удобен и в том случае, когда расширение обобщения желательно скрыть от клиентского модуля, который может использовать только предоставляемые ему специализации, формируемые в других модулях.

В отличие от концепции базового типа, расширяемого за счет добавления в производных типах, использование параметрического обобщения предполагает

сохранение внешнего восприятия, а альтернативные трактовки вводятся как уточнения исходного типа. Таким образом, внешне все специализации имеют единый тип, а их разнообразное толкование используется только внутри него. Это позволяет убрать глобальную идентификацию типов, применяемую при расширении записей или наследовании, и обеспечивает поддержку концепции строгой типизации. Использование локальной идентификации альтернатив в свою очередь позволяет реализовать табличный доступ к обработчикам специализаций обобщающих процедур, что значительно ускоряет их вызов.

В качестве альтернативы расширению записей, приведенному в [12], рассмотрим, как можно реализовать аналогичное решение с применением обобщенных записей. Пусть базовый тип будет выстроен как запись с пустым обобщением:

```
T = RECORD x, y: INTEGER; CASE END;
```

Тогда от этой записи можно независимо выстроить две специализации с явным указанием признаков:

```
T += t0: BOOLEAN;
T += t1: RECORD r: REAL; s: CHAR END;
```

Используя построенную обобщенную запись можно получить переменные типа ***T***, с двумя специализациями ***t0*** и ***t1***:

```
v0: T(t0); v1:T(t1);
```

В качестве примера можно привести следующие варианты доступа к полям этих переменных:

```
v0.x, v1.y, v0(), v1().r
```
…

Следует также отметить, что круглые скобки, помимо задания признаков, отделяют поля основной записи от полей специализаций, что позволяет использовать в обоих понятиях одинаковые имена.

3. РЕАЛИЗАЦИЯ ОБОБЩЕННЫХ ЗАПИСЕЙ

Реализация программных объектов в современных языках программирования является большим, чем простое отображение структур данных на память используемого вычислителя. Через дополнительные структуры данных осуществляется инструментальная поддержка новых свойств, что обеспечивает гибкое расширение программы и повторное использование разработанных артефактов. В частности, в языке программирования C++ [13] в классы включена таблица виртуальных методов, обеспечивающая полиморфизм и, при необходимости, идентификацию типов во время выполнения программы. Динамическая идентификация типов в языке программирования Оберон [14] поддерживается за счет включения в расширяемую запись идентификатора типа. Полиморфизм типа, обеспечивающий динамическую проверку типов во время выполнения и их связь с базовым типом, поддерживается за счет использования в расширяемых записях указателей на родительский тип [12]. При этом программные объекты, расширяющие родительский тип, и в том и другом случае рассматриваются как новые типы, а идентифицирующие их признаки используются в глобальном контексте.

Отличие обобщенной записи проявляется в инструментальной поддержке механизма идентификации, позволяющего рассматривать специализации не как новые типы данных, а как частные расширения уже существующего типа, задающего основу обобщенной записи, имеющей следующий формат:

ОсноваОбобщеннойЗаписи = ПоляОбычнойЗаписи ПризнакСпециализации.

Признак специализации – это дополнительное поле, значение которого устанавливается в зависимости от подключаемой специализации.

Специализированные записи дополнительно имеют поле, содержащее основу специализации, которое располагается после ее признака:

СпециализированнаяЗапись = ОснованаОбобщеннойЗаписи ОснованаСпециализации.

Использование для формирования обобщений рассмотренного выше формата позволяет локализовать пространство признаков, сопоставив с каждой из подключаемых специализаций число в диапазоне от 0 до n, где n – количество подключенных специализаций. Локализация контекста, в свою очередь облегчает анализ и обработку специализаций, обеспечивая при этом более простую реализации операций, поддерживающих полиморфизм типов и процедур.

В качестве примера можно рассмотреть генерацию кода для следующего описания, определяющего обобщенную запись:

```
T = RECORD x, y: INTEGER; CASE END;
```

Для этой строки компилятор сгенерирует код, аналог которого на языке программирования C++ может быть представлен следующим образом:

```
struct T {
   int x, y; // Поля записи
   int rank; // Поле признака специализации
};
```

Компилятор также порождает дополнительные структуры, обеспечивающие регистрацию признаков предполагаемых специализаций и хранения структур специализаций, упорядоченных в соответствии со значениями их признаков. Помимо этого для данной обобщенной записи присваивается нулевой ранг, как записи, формируемой по умолчанию.

В том случае, если основы специализации при расширении представляются в виде ранее не определявшихся типов, для каждого из расширений обобщения осуществляется генерация соответствующей специализированной записи, которая регистрируется в массиве специализаций. В частности, для специализаций:

```
T += t0: BOOLEAN;
T += t1: RECORD r: REAL; s: CHAR END;
```

будут построены объекты, соответствующие следующим структурам, языка C++:

```
// для T += t0: BOOLEAN;
struct T_t0 { // для признака t0
   int x, y; // Поля записи
   int rank; // Поле признака специализации
   bool _t0; // добавление булева поля
};

// для T += t1: RECORD r: REAL; s: CHAR END;
struct T_t1 { // для признака t0
   int x, y; // Поля записи
   int rank; // Поле признака специализации
   double r_t1; // добавление поля r
   char s_t1; // добавление поля s
};
```

Значения признаков специализаций для каждой из сформированной структур задаются в последовательности их обработки компилятором или во время окончательной компоновки, если добавление специализаций осуществляется в различных единицах компиляции. Инструментальная поддержка обеспечивает одинаковое для всех специализированных записей толкование основы, анализ признака специализации и автоматическое определение по этому признаку подключенной основы специализации. Во время создания переменной с заданной специализацией поле признака инициализируется соответствующим его значением.

Это позволяет легко определить тип специализированной записи во время выполнения и осуществить обработку артефакта в соответствии с идентифицированной структурой.

Аналогичным образом осуществляется генерация кода и для других вариантов построения обобщенных записей. Параметрическое обобщение в данном случае будет отличаться от обобщенной записи только отсутствием полей данных, соответствующих полям обычной записи.

4. ОСНОВНЫЕ ОПЕРАЦИИ НАД ОБОБЩЕННЫМИ ЗАПИСЯМИ

Базовые операции обработки обобщенных записей практически не отличаются от соответствующих операций обработки параметрических обобщений, представленных в [10, 11]. В основе их лежат операции обработки расширяемых записей языка программирования Оберон [12, 14]. В частности, допускается статическое и динамическое создание специализированных записей, указателей на обобщенные и специализированные записи, явное приведение обобщенного типа к типу специализации, проверка типа специализации подключенной к указателю на обобщенную запись.

Проверка специализации осуществляется операцией **IS**, в которой первым операндом является указатель на обобщенную запись, а в качестве второго операнда выступает признак специализации. Если к указателю на обобщенную запись подключена проверяемая специализированная запись, то в качестве результата возвращается значение **TRUE**. Во всех иных случаях возвращается **FALSE**. Пример использования:

```
VAR pv: POINTER T; …
pv := NEW(T(t0)); …
IF pv IS T(t0) THEN … ELSE …
```

Прямое преобразование типа специализации может также применяться к указателю на обобщенную запись. Преобразование успешно завершается, если подключаемая специализация соответствует преобразуемому типу. В противном случае происходит аварийное завершение программы. Данная операция должна использоваться совместно с операцией **IS**. Например:

```
VAR pv: POINTER T; v: T(t0);
v := NEW(T(t0)); …
IF v IS T(t0) THEN v() := pv(t0) END; …
```

В примере осуществляется присваивание значения обобщенной части динамической переменной обобщенному полю специализированной переменной. Предварительная проверка типа позволяет определить, что динамическая переменная ***pv*** имеет специализацию ***t0***. В противном случае присваивание не осуществляется.

Обобщенную запись и ее специализации допускается использовать в операторах присваивания. В частности, при наличии эквивалентных специализаций в левой и правой частях операторов присваивания, осуществляется присваивание всех полей записи, стоящей справа от знака присваивания, полям, расположенным в его левой части. Если специализации в левой и правой частях различны, то присваивание осуществляется только для общих полей обычной записи. Допускается также прямое присвоение полям специализации обобщенной записи полей основы специализации. Аналогичная ситуация возможна и при использовании в левой части оператора присваивания основы специализации, когда в правой его части располагается соответствующая специализированная запись. В этом случае поля основы заполняются соответствующими полями специализации.

5. ИСПОЛЬЗОВАНИЕ ОБОБЩАЮЩИХ ПАРАМЕТРИЧЕСКИХ ПРОЦЕДУР

Обобщенные записи могут использоваться в качестве параметров в обобщающих параметрических процедурах аналогично тому, как используются обобщения [10, 11]. Описание обобщающей параметрической процедуры имеет следующий вид:

ОбобщающаяПроцедура = PROCEDURE Имя
СписокОбобщающихПараметров [ФормальныеПараметры]
((ТелоПроцедуры Имя) | ":= 0") .

Отличие от обычной процедуры заключается в присутствии списка обобщающих параметров:

СписокОбобщающихПараметров = "{"ГруппаОбобщающих
{";" ГруппаОбобщающих } "}".
ГруппаОбобщающих = [VAR] Идентификатор
{ "," Идентификатор }":" ОбобщающийТип.

Тело содержит обработчик по умолчанию, если для всех обобщающих параметров существует тип по умолчанию. В противном случае оно содержит обработчик исключений. Тело обобщающей процедуры может отсутствовать, что задается «приравниванием» его нулевому значению (по аналогии с чистыми функциями языка программирования C++). В этом случае необходимы обработчики специализаций для различных комбинаций обобщенных параметров.

Обобщающая параметрическая процедура для вычисления периметра любой геометрической фигуры типа **Figure** выглядит следующим образом:

```
(* Если нужен только общий интерфейс *)
PROCEDURE P {VAR s: Figure}: REAL := 0

(* Если используется обработчик по умолчанию *)
TYPE PFigure = POINTER TO Figure; …
PROCEDURE P2 {ps: PFigure}; BEGIN
  SendException('Incorrect parameter')
END P2
```

Обработчики обеспечивают реализацию различных комбинаций специализаций, сопоставляемых с обобщениями из списка обобщающих параметров. Комбинация, на которую «настроен» конкретный обработчик, задается значениями признаков в соответствии со следующими синтаксическими правилами:

ОбработчикСпециализации = PROCEDURE Имя
СписокСпециализаций [ФормальныеПараметры]
ТелоПроцедуры Имя .
СписокСпециализаций = "{" ГруппаСпециализаций
{ ";" ГруппаСпециализаций } "}" .
ГруппаСпециализаций = [VAR] Идентификатор { "," Идентификатор }
":" ОбобщающийТип "(" [Признак] ")"
| [VAR] Идентификатор "(" [Признак] ")"
{ "," Идентификатор "(" [Признак] ")"} ":"
ОбобщающийТип.

Каждый элемент списка специализированных параметров должен задавать конкретное значение признака. Специализации должны поэлементно соответствовать параметрам обобщающей процедуры. В обязательном теле процедуры кодируется конкретный метод обработки. Можно использовать один из двух способов задания специализаций, более удобный в рассматриваемом контексте: несколько одинаковых специализаций в группе или несколько разных специализаций в группе. Для представленных выше примеров обобщенных процедур допустимы следующие обработчики, вычисляющие периметры конкретных геометрических фигур:

```
(* Вычисление периметра прямоугольника *)
PROCEDURE P {VAR r: Figure(rect)}: INTEGER;
BEGIN RETURN 2*(r().x + r().y) END P;

 (* Вычисление периметра треугольника *)
PROCEDURE P {VAR t(trian): Figure}: INTEGER;
BEGIN RETURN t().a + t().b + t().c END P;
```

Следует отметить, что для обеспечения доступа к полям специализаций необходимо перед именами полей указывать круглые скобки. Это обеспечивает их отличие от полей с аналогичными именами, встречающимися в основной записи.

6. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Построение новых специализаций может также осуществляться на основе обобщенной записи, что позволяет формировать цепочки уточнений произвольной длины. При этом следует отметить, что подобное введение новых специализаций возможно только в том случае, если предшествующий тип является обобщенным. Это позволяет контролировать процесс добавления новых уточнений. Например, для формирования новой ступени обобщения ***T*** необходимо включить в него, в качестве специализации первого уровня обобщение ***T0***, содержащего свою «точку» для расширения, которую можно «подключить» к ***T***:

```
T0 = RECORD z: INTEGER; CASE OF END;
T += CASE OF t2: T0; END;
```

Тип T0 можно будет уточнять, добавляя для этого к нему новые специализации, которые также могут содержать обобщения, обеспечивающие их дальнейшее уточнение:

```
T00 = RECORD a: INTEGER; CASE OF END;
T0 += t00: T00;
```

Использование данного приема позволяет выстраивать сложные зависимости между типами, воспринимая при этом различные специализации как уточнения одного и того же типа. Для приведенного примера могут быть сформированы следующие специализации:

из **T → T(t0)** или **T(t1)** или **T(t2)**,
из **T(t2) → T(t2).(t00)** и т.д.

Допускается также рекурсивное подключение к существующим обобщениям других обобщений, включая и подключение самого себя. При необходимости это позволяет выстраивать длинные статические цепочки, формируемые во время компиляции программы. В качестве примера можно расширить тип T специализацией, построенной на основе этого же типа:

```
T += t3: T;
```

Тогда появляется возможность выстраивать следующие специализации:

```
T(t3), T(t3).(t3), T(t3).(t3).(t3).(t3).(t2).(t00), …
```

7. ОТЛИЧИЕ ОТ СУЩЕСТВУЮЩИХ МЕТОДОВ СОЗДАНИЯ ОБОБЩЕНИЙ

Формирование обобщений связано с постоянным поиском баланса между возможностями надежного контроля их внутренней структуры и гибкостью наращивания. Последнее обуславливается тем, что механизм альтернативной подмены программных объектов широко используется не только для реализации новых свойств уже существующих понятий, но и для прямого расширения ранее

существующей функциональности. Поэтому, представляет интерес сравнение возможностей обобщенных записей, используемых в ППП, с вариантами построения обобщений, применяемыми в других парадигмах.

К одному из наиболее простых методов формирования обобщений следует отнести подключение произвольных структур данных к общему указателю. Этот подход широко применяется в языке программирования С [15]. За возможность подключения чего угодно и к чему угодно приходится платить полным отсутствием контроля типов во время компиляции. Вся ответственность за правильное использование обобщений возлагается на программиста. В частности, обобщение ***V***, построенное с использованием альтернативных структур данных ***T1***, ***T2***, ***T3*** в качестве специализаций, может выглядеть следующим образом:

```
// ключи, идентифицирующие специализации
enum keyType {kT1, kT2, kT3};
Struct V {
  keyType k; // ключ специализации
  void *t; // указатель на любую подключаемую альтернативу
};
```

В самой программе для создания трех альтернативных объектов необходимо писать специальный код, обеспечивающий создание экземпляров специализаций и самих обобщений. Например, при использовании статического размещения альтернатив, обобщения можно сформировать следующим образом:

```
T1 t1; T2 t2; T3 t3; // основы альтернативных специализаций
V v1 = {kT1, (void*)&t1};   // специализация v1 на основе t1
V v2 = {kT1, (void*)&t2};   // специализация v2 на основе t2
V v3 = {kT1, (void*)&t3};   // специализация v3 на основе t3
```

В том же языке С можно использовать объединение union, которое позволяет налагать множество альтернативных специализаций на одно и то же пространство памяти. В этом случае построение обобщений упрощается:

```
Struct V {
  keyType k; // ключ специализации
  union { // объединение альтернатив
    T1 t1;
    T2 t2;
    T3 t3;
  }
};
```

Следует отметить, что по уровню контроля за правильным использованием специализаций данный подход мало отличается от предыдущего. В частности, для того, чтобы проверить тип специализации, загруженной в объединение, приходится использовать дополнительные переменные. К недостатку следует отнести и то, что любое расширение объединения ведет к модификации его структуры. Кроме того, размер объединения определяется размером его максимальной специализации, что может вести к неэффективному использованию памяти при обобщении разнотипных структур.

Более надежная, по сравнению с С, организация обобщения использована в языках Паскаль [16] и Модула-2 [17]. Для этого в них введена вариантная запись, содержащая признак специализации, который явно задается программистом. Это обеспечивает более строгий контроль зависимости между специализацией и признаками, а также дисциплинирует программиста при обработке вариантных записей в операторах выбора. Вместе с тем следует отметить отсутствие динамического контроля правильности использования специализаций в зависимости от признака, что снижает надежность программирования. Программист может явно изменить признак вне зависимости от хранимых данных. Помимо этого он может обращаться к полям произвольной специализации независимо от того, на какую специализацию указывает признак. К традиционному

для процедурного подхода недостатку относится невозможность безболезненного расширения структуры вариантной записи.

Динамический контроль вариантов используется в объединениях языка программирования Ада [18]. В нем оператор присваивания устанавливает значение объекта соответствующего типа, которое впоследствии не может трактоваться по-иному до выполнения следующего присваивания. Попытка обратиться к объединению как к элементу другого типа ведет к ошибке периода выполнения. Подобный контроль повышает надежность программирования. Вместе с тем следует отметить, что это решение не обеспечивает безболезненного эволюционного расширения программы. Помимо этого пространство признаков непосредственно ассоциируется с типами данных, определяющих специализации, что ведет к необходимости обработки глобального контекста (следует отметить, что в рассматриваемом контексте глобальные признаки можно легко свести к локальному контексту, поэтому данное замечание не является существенным).

Проблемы расширения программ, возникающие в связи с использованием специализаций, размещаемых внутри обобщений, привели к поиску конструкций, допускающих внешнее подключение специализаций, но с более надежной проверкой типов, чем при использовании произвольных структур, допускаемых в языке С. Одним из удачных решений в данном направлении является использование механизма наследования, при котором специализация образуется как расширение базового понятия. Наличие аналогичной основы в обобщении и специализациях позволяет осуществлять динамическую подмену. Этот подход широко используется в современных языках ОО программирования, например, в C++, Java, C#. В сочетании с механизмом виртуализации он обеспечивает во многих ситуациях эволюционно расширяемую разработку программ. К недостаткам подобных конструкций можно отнести невозможность прямого построения новых обобщений от уже существующих специализаций. Помимо этого следует отметить и то, что использование виртуализации и наследования обеспечивает эволюционное расширение только для монометодов [4]. Возникают проблемы с эволюционной реализацией мультиметодов, которые могут применяться при решении многих задач. Это ведет к дополнительному использованию явной проверки типов объектов во время выполнения, что наделяет ОО программы процедурными свойствами и затрудняет их эволюционное расширение. К определенным проблемам, обуславливаемой размещением процедур внутри данных при использовании ОО подхода, является и добавление новых процедур.

Построение специализаций от обобщения на основе наследования возможно и в процедурном программировании. Инструментальная поддержка этого метода для расширения типов данных предложена в работе [12] и реализована в языке программирования Оберон [14]. Использование данного механизма обеспечило более жесткий контроль типов данных, расширяющих обобщение, который, однако, был перенесен на период выполнения программы. Вместе с тем, при построении сложных структур данных и их анализа стало необходимым использовать глобальную информацию о типах данных, которая трудно поддается параметризации во время компиляции. Это ведет к явному анализу типов специализаций во время выполнения программы и невозможности безболезненно расширить уже написанный код, связанный с обработкой альтернатив обобщения.

ЗАКЛЮЧЕНИЕ

Применение параметрических обобщений в сочетании с обобщающими параметрическими процедурами привело к более гибкому созданию эволюционно расширяемых программ по сравнению с ранее используемыми подходами. Использование обобщенных записей, вместо параметрических обобщений, ведет к дополнительному повышению гибкости при формировании структур данных в процедурно-параметрических программах. Не внося кардинальных изменений в

ППП, эти записи позволяют формировать обобщения без создания дополнительных промежуточных структур, которые приходилось использовать до этого. Трудоемкость построения типов данных становится соизмеримой с созданием расширяемых записей языка программирования Оберон или иерархий классов объектно-ориентированных языков.

СПИСОК ЛИТЕРАТУРЫ

[1] **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Пер. с англ. - СПб: Питер, 2001. - 368 с.

[2] **Мейерс С.** Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов: Пер. с англ. - М.: ДМК Пресс, 2000. - 304 с.

[3] **Легалов А.И.** ООП, мультиметоды и пирамидальная эволюция. *// Открытые системы* - 2002, № 3 (март). С. 41-45.

[4] **Легалов А.И.** Мультиметоды и парадигмы. *// Открытые системы* - 2002, № 5 (май). С. 33-37.

[5] **Вандэвурд Д. Д., Джосаттис Н. М.** Шаблоны C++: справочник разработчика. : Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 544 с.

[6] **Steele G.L.** Common Lisp the Language, 2nd Edition. – Digital Press, Bedford, Massachusetts, 1990 (электронная версия документа размещена на сервере http://www.cs.cmu.edu/).

[7] **Страуструп Б.** Дизайн и эволюция C++: Пер. с англ. - М.: ДМК Пресс, 2000. - 448 с.

[8] **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.

[9] **Легалов А.И.** Методы поддержки параметрического полиморфизма // Научный вестник НГТУ. – 2004. – № 3 (18). – С. 73-82.

[10] **Легалов А.И. Швец Д.А.** Язык программирования O2M (электронная версия документа размещена по адресу http://www.softcraft.ru/ppp/o2m/o2mref.shtml).

[11] **Легалов А.И. Швец Д.А.** Процедурный язык с поддержкой эволюционного проектирования. – Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.

[12] **Wirth N.** Type Extensions. / ACM Transactions on Programming Languages and Systems. Vol. 10, No 2, April 1988, p. 204-214.

[13] **Страуструп Б.** Язык программирования C++. Третье издание. /Пер. с англ. - СПб.; М.: "Невский диалект" - "Издательство БИНОМ", 1999. - 991 с.

[14] **Reiser M., Wirth N.** Programming in Oberon: steps beyond Pascal and Modula. / Addison-Wesley, ACM Press. – 1992.

[15] **Джехани Н.** Программирование на языке Си: Пер. с англ. – М.: Радио и связь, 1988. – 272 с.

[16] **Немюгин С., Перколаб Л.** Изучаем Turbo Pascal. – СПб., Питер. 2000.

[17] **Вирт Н.** Программирование на языке Модула-2. /Пер. с англ. - М.: Мир, 1987. – 244 с.

[18] **Джехани Н.** Язык Ада. /Пер. с англ. – М.: Мир, 1988. – 522 с.