УДК 004.432.2

А. И. Легалов, А. Ф. Солоха

Сибирский федеральный университет
пр. Свободный, 79, Красноярск, 660041, Россия
E-mail: b0ntrict0r@yandex.ru; legalov@mail.ru

ОСОБЕННОСТИ ЯЗЫКА ПРОЦЕДУРНО-ПАРАМЕТРИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Приводится ряд специфических особенностей языка процедурно-параметрического программирования. Реализованная процедурно-параметрическая парадигма позволяет преодолеть ряд недостатков, присущих объектно-ориентированному подходу, обеспечивая эффективную эволюционную поддержку динамического множественного полиморфизма, применяемого в мультиметодах. Для этого предлагаются обобщенные записи и параметрические процедуры. Для повышения гибкости при расширении программ используется новая модульная структура.

Ключевые слова: языки программирования, парадигмы программирования, эволюционная разработка программ, модульное программирование, процедурно-параметрическое программирование.

Введение

Эволюционная разработка больших программ связана не только с методами их проектирования, большую роль играет использование соответствующих языков программирования. Доминирующий в настоящее время объектно-ориентированный (ОО) подход обеспечивает частичную инструментальную поддержку безболезненному расширению кода за счет использования механизмов наследования и виртуализации. Вместе с тем при сложных взаимодействиях между классами эволюционное расширение программы только за счет ОО-программирования становится затруднительным. Приходится использовать дополнительные алгоритмические приемы и динамическое связывание объектов, что, например, нашло широкое отражение в образцах (паттернах) проектирования [1]. Часто только ОО-парадигмы бывает недостаточно для расширения программы, поэтому приходится переходить к мультипарадигменному стилю [2], базирующемуся на сочетании различных подходов. Необходимость использования сочетания парадигм и подходов во многом обусловлена существованием и таких задач, при решении которых применяется множественный полиморфизм, т. е. мультиметоды. И хотя в данной ситуации возможно чистое объектно-ориентированное эволюционное решение, опирающееся на диспетчеризацию [3], более эффективными являются мультипарадигменные варианты [3–5].

Вместе с тем следует отметить отсутствие в настоящее время языковых средств, обеспечивающих эффективную поддержку мультиметодов, что не позволяет использовать их напрямую. Одной из широко известных реализаций является включение мультиметодов в язык CLOS [6]. Однако предложенные решения обладают невысокой производительностью. Другим подходом, изначально ориентированным на эффективную поддержку мультиметодов, является процедурно-параметрическая (ПП) парадигма программирования [7], обеспечивающая инструментальную поддержку эволюционной разработки программ в рамках процедурного стиля. Для его исследования разработан язык программирования O2M [8]. В последующих работах были представлены результаты, расширяющие возможности парадигмы за счет использования обобщенных записей [9] и подключаемых модулей [10].

Появление новых конструкций привело к ряду новых возможностей, расширяющих процедурное программирование, которые были включены в язык процедурно-параметрического

программирования Alien. Как и O2M, новый язык использует синтаксис, аналогичный синтаксису языка программирования Оберон-2 [11]. Однако если O2M разрабатывался как расширение Оберона-2, то Alien является языком, ориентированным на поддержку только процедурного подхода. Это обусловлено тем, что предлагаемые ПП понятия полностью дублируют, а зачастую и перекрывают имеющиеся в Обероне-2 расширяемые типы данных и процедуры, связанные с типом, обеспечивающие объектно-ориентированное программирование. Использование обобщающих процедур в сочетании с обобщенными записями обеспечивает поддержку множественного полиморфизма и включает одиночный полиморфизм, используемый в ОО-подходе в качестве частного случая. Кроме того, для исследования возможностей ПП-подхода и новых методов модульной организации целесообразным является создание языка, ориентированного только на эту парадигму.

Обобщенные записи

Обобщенная запись развивает концепцию процедурно-параметрического обобщения, добавляя в нее дополнительные поля, общие для всех специализаций. Это обеспечивает сходство обобщенной записи с обычной записью таких языков программирования, как Паскаль, Модула-2, Оберон, а также позволяет использовать эти записи в обобщающих параметрических процедурах. Разработка данного понятия затрагивает как его внутреннюю структуру, используемую для создания переменных, так и синтаксис языка программирования, который в рамках проводимых работ должен быть дополнен новыми правилами. Синтаксис обобщенной записи выглядит следующим образом:

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

Параметрическое обобщение располагается в конце записи и позволяет использовать общие поля во всех специализациях, добавляемых при расширении. Допустимость только одного обобщения позволяет применять признак специализации для характеристики всей записи. В примере

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

T0 – обобщенная запись, которая содержит пустое обобщение. В следующей строке к записи добавляется специализация с признаком *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 альтернативное обозначение:

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

Скобки при этом остаются, так как они сигнализируют об использовании обобщенной части.

Допускается создавать записи с уже существующими обобщениями. В качестве примера можно рассмотреть обобщение геометрической фигуры с добавлением к каждой из конкретных фигур целочисленного поля, информирующего, например, о ее цвете:

```
ColoredFigure = RECORD color: INTEGER; CASE Figure END;
```

где *Figure* – ранее разработанное обобщение. Подход удобен, когда расширение обобщения желательно скрыть от импортирующего 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 исходного типа, а альтернативные специализации вводятся как его уточнения. 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 вместо алгоритмического табличный доступ к обработчикам специализаций обобщающих процедур, что значительно ускоряет их вызов.

В качестве альтернативы расширяемым записям [11] можно реализовать аналогичное решение с применением обобщенных записей. Пусть базовый тип будет выстроен как запись с пустым обобщением:

```
T = RECORD x, y: INTEGER; CASE OF 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 отметить, что круглые скобки, помимо задания признаков, отделяют поля основной записи от полей специализаций, что позволяет использовать в обеих частях обобщенной записи одинаковые имена.

Базовые операции над обобщенными записями

Базовые операции обработки обобщенных записей практически не отличаются от соответствующих операций обработки параметрических обобщений, представленных в [8]. В их основе лежат операции обработки расширяемых записей языка программирования Оберон-2 [11]. Допускается статическое и динамическое создание специализированных записей, указателей на обобщенные и специализированные записи, явное приведение обобщенного типа к типу специализации, проверка типа.

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 специали-

зированная запись, то в качестве результата возвращается значение TRUE. Во всех иных случаях возвращается FALSE. Пример использования:

```
VAR pv: POINTER TO T; …
NEW(pv(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 TO T; v: T(t0);
NEW(pv(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. При наличии эквивалентных специализаций в левой и правой частях операторов присваивания осуществляется присваивание всех полей записи, стоящей справа от знака «:=», полям, расположенным в его левой части. Если специализации различны, то присваивание осуществляется только для общих полей обычной записи. Допускается также прямое присваивание 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.

Использование обобщающих параметрических процедур

Обобщенные записи могут использоваться в качестве параметров в обобщающих параметрических процедурах аналогично тому, как используются параметрические обобщения [8]. 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.

Дополнительные возможности обобщенных записей

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,

когда предшествующий тип является обобщенным. Это позволяет контролировать процесс добавления новых уточнений. Например, для формирования новой ступени обобщения T необходимо включить в него обобщение T0, содержащее свою «точку» для расширения, которую можно «подключить» к T:

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

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. 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), …
```

Использование обобщенной записи позволяет применять технику формирования структур данных, базирующуюся на параметрическом подключении в ее конец другой обобщенной записи. Так, можно набирать длинные статические цепочки требуемой конфигурации из универсальных базовых конструкций. Аналогичные по структуре декорирующие элементы используются в образце ОО проектирования «Декоратор» [1], формируя окончательную структуру путем динамического связывания во время выполнения программы. Опираясь на эти элементы, задаваемые обобщенными записями, можно собирать из них разнообразные типы. Один и тот же декорирующий тип может многократно использоваться при порождении нового типа. Например, еще один вариант цветной геометрической фигуры может быть построен из небольших элементов следующим образом:

```
// Декоратор Точки
PointDecorator = RECORD p:Point CASE Decorator END;
// Декоратор Цвета
ColorDecorator = RECORD c:Color CASE Decorator END;
// Декоратор угла
AngleDecorator = RECORD alpha:REAL; CASE Decorator END;
// Общий декоратор
Decorator = CASE TYPE OF
PointDecorator | ColorDecorator |
AngleDecorator | Figure
END;
// Цветная фигура на плоскости через декораторы
NewFigure = PointDecorator(AngleDecorator(ColorDecorator(
```

```
ColorDecorator(Figure))));
VAR      circle: NewFigure(Circle);
rectangle: NewFigure(Rectangle);
```

Подобные приемы повышают гибкость ПП-парадигмы и в сочетании с множественным полиморфизмом обобщающих процедур обеспечивают эффективное создание эволюционно расширяемых и повторно используемых программных объектов.

Подключаемые модули

Идея подключаемых модулей схожа с использованием механизма наследования вместо прямого включения типов во вновь формируемый программный объект. В случае с наследованием формируется описание нового типа, расширяющего базовый тип дополнительными понятиями. За счет принципа подстановки осуществляется использование метода производного класса, который может обрабатывать свои собственные дополнительные данные. При использовании подключаемых модулей ситуация отличается тем, что вместо множества экземпляров базового и производного классов в программе существует только по одному экземпляру разных модулей. Поэтому данный метод не является аналогом наследования. Вместо этого разработанное расширение модуля может подключаться к уже существующему базовому модулю, образуя вместе с ним единое пространство имен. Это отличает подключение модуля от его импорта, при котором внутренние пространства имен модулей не пересекаются. Подключаемый модуль может импортироваться из других модулей, обеспечивая им передачу своего интерфейса и интерфейса расширяемого модуля. Главной особенностью подключаемого модуля является возможность описания в нем расширений обобщенных записей и обобщающих параметрических процедур.

Синтаксис подключаемого модуля имеет следующий вид:

```
ПодключаемыйМодуль =
MODULE идент "("идент" ["*"|"+" "])"";" [СписокИмпорта]
ПоследовательностьОбъявлений
[BEGIN ПоследовательностьОператоров] END идент ".".
СписокИмпорта = IMPORT Импорт {"," Импорт} ";".
Импорт = [идент ":="] идент.
```

Отличие от обычного модуля проявляется в указании имени модуля-родителя (базового модуля) в круглых скобках. Знак «*» или «+» за именем родителя определяет права доступа к интерфейсу родительского модуля из модулей, импортирующих подключаемый модуль. Правила использования этих знаков совпадают с их применением в языке программирования Оберон.

На практике это позволяет организовать прямые взаимодействия, отличающиеся от существующих языковых структур, используемых в языках Оберон, Оберон-2, O2M. В [10] рассмотрен пример программы, демонстрирующий гибкое добавление новых геометрических фигур и их обобщающих процедур, включая и мультиметоды.

Заключение

Разработка языка программирования Alien позволяет провести исследование возможностей процедурно-параметрической парадигмы и наметить дальнейшие пути ее развития. В частности, инструментальная поддержка эволюционного расширения мультиметодов позволяет гибко и без дополнительных алгоритмических затрат осуществлять прямое написание кода и для тех ситуаций, когда мультиметоды вырождаются в более простые конструкции. Многие из таких конструкций в настоящее время реализованы с применением паттернов проектирования. ПП-парадигма в целом позволяет многие из паттернов описывать напрямую, т. е. без задания сложных отношений между классами и без динамического связывания

во время выполнения. Это повышает эффективность выполнения написанного кода и снижает его размер.

Другой специфической особенностью является то, что процедурно-параметрическая парадигма хорошо согласуется с функциональной, обеспечивая при этом дополнительные возможности по эволюционному расширению программ. Работы в этом направлении позволяют создавать архитектурно-независимые параллельные языки, обеспечивающие дополнительные возможности гибкого сопровождения и развития программного обеспечения.

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. *Коплиен Дж.* Мультипарадигменное проектирование для C++. Библиотека программиста. СПб.: Питер, 2005. 235 с.

3. *Легалов А. И.* ООП, мультиметоды и пирамидальная эволюция // Открытые системы. 2002. № 3. С. 41–45.

4. *Легалов А. И.* Мультиметоды и парадигмы // Открытые системы. 2002. № 5. С. 33–37.

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

6. *DeMichiel L. G., Gabriel R. P.* The Common Lisp Object System: An Overview. URL: http://www.dreamsongs.com/NewFiles/ECOOP.pdf

7. *Легалов А. И.* Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? Красноярск, 2000. Рук. деп. в ВИНИТИ 13.03.2000. № 622-В00. 43 с.

8. *Легалов А. И., Швец Д. А.* Язык программирования O2M. URL: http://www.soft-craft.ru/ppp/o2m/o2mref.shtml

9. *Легалов И. А.* Применение обобщенных записей в процедурно-параметрическом языке программирования // Науч. вестн. НГТУ. 2007. № 3 (28). С. 25–38.

10. *Легалов А. И., Бовкун А. Я., Легалов И. А.* Расширение модульной структуры программы за счет подключаемых модулей / Докл. АН ВШ РФ. 2010. № 1 (14). С. 114–125.

11. *Свердлов С. З.* Языки программирования и методы трансляции: Учеб. пособие. СПб.: Питер, 2007. 638 с.

Материал поступил в редколлегию 26.07.2011

A. I. Legalov, A. F. Soloha

THE FEATURES OF PROCEDURE-PARAMETRIC PROGRAMMING LANGUAGE

Some specific opportunities of procedure-parametric programming language described. The procedural-parametric programming paradigm overcomes a number of shortcomings specific to a pure object-oriented design and programming approach. It supports more effective evolution software development using multimethods. Generalized records and parametric procedures proposed for it. Furthermore new module organization was used for flexible extension of programs.

Keywords: programming languages, programming paradigms, evolution software development, module programming, procedural-parametric programming.