УДК 681.3.06

ЭВОЛЮЦИОННОЕ РАСШИРЕНИЕ ПРОГРАММ В ФУНКЦИОНАЛЬНОМ ЯЗЫКЕ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ[1]

А.И. Легалов*, Д.В. Привалихин**

Рассматриваются программные объекты функционального языка параллельного программирования «Пифагор», обеспечивающие построение эволюционно расширяемых программ. Для достижения заданного эффекта используются перегрузка функций и динамические типы данных, задаваемые пользователем.

Язык программирования Пифагор [1] предназначен для разработки параллельных программ, управление вычислениями в которых осуществляется по готовности данных. Существующие на данный момент версии языка и исполнительной системы [2] поддерживают только динамическую типизацию. Формирование новых типов данных осуществляется неявно в ходе выполнения функций. Большинство же современных языков данной группы поддерживают строгую типизацию.

Вместе с тем механизмы динамической типизации могут развиваться по специфичным для них направлениям, что позволяет получать интересные эффекты, расширяющие возможности языка. В частности, сочетание механизма перегрузки функций и динамических типов, определяемых пользователем, обеспечивает написание эволюционно расширяемых параллельных программ.

Перегрузка функций с одинаковой сигнатурой

В языке отсутствует строгая типизация, а все функции с одинаковыми именами неразличимы (имеют одинаковую сигнатуру). Поэтому, вместо выбора одной из перегруженных функций, осуществляется их одновременное выполнение. Результат возвращается в виде параллельного списка. В отличие от обычных функций, перегруженные функции задаются добавлением к обозначению имени квадратных скобок, в которых может стоять любое действительное число, определяющее ранг.

Перегруженная_функция := имя_функции "[" ранг "]" <<
"funcdef" [аргумент] "{" [элемент {";" элемент }] "}".

Ранг используется для упорядочения функций в параллельном списке по возрастанию. При отсутствии числа ранг считается равным нулю. Функции с одинаковым рангом могут располагаться в произвольной последовательности. Обычно они размещаются в порядке их обработки транслятором. Пример использования рангов:

```
OverFunc[2.5] << funcdef Param { // Тело функции }
OverFunc[]    << funcdef Param { // Тело функции }
OverFunc[-10] << funcdef { // Тело функции }
```

Вызов параллельной функции синтаксически ничем не отличается от обычного вызова:

```
X:OverFunc;
```

---

[1] При поддержке РФФИ № 02-07-90135.

Перегрузка функций позволяет гибко добавлять новые возможности, обусловливаемые появлением дополнительной информации о разрабатываемом алгоритме. В частности, можно использовать методы, обеспечивающие децентрализованную разработку отдельных фрагментов функций, связанных с вычислением при альтернативных условиях. Подобный прием децентрализованной разработки функций используется во многих функциональных языках, в частности, ML [3] и Haskel [4]. Однако в них децентрализованное описание альтернатив ограничивается заданием статических типов на этапе компиляции. Проверки значений величин, осуществляемые в ходе выполнения программы, напрямую не связываются с типами. Это ограничивает возможности использования децентрализованного описания и не позволяет расширять функции, в которых типы и значения данных определяются в ходе вычислений.

В качестве простейшего примера перегрузки функций с одинаковой сигнатурой можно рассмотреть вычисление факториала. Пусть факториал отдельно вычисляется для значений аргумента, равного 0, 1, диапазона от 2 до **n** (где **n** – константа, определяющая максимально допустимую величину аргумента). При значении аргумента, превышающем n, выводится сообщение о переполнении. При отрицательном значении аргумента выводится сообщение об ошибке. Константа **n** определена в программе следующим образом:

```
n << const 10;
```

Функция вычисления факториала разделяется на подфункции, каждая из которых производит необходимые вычисления для аргумента, расположенного в ее диапазоне. Если значение аргумента не попадает в диапазон, обрабатываемый конкретной перегруженной функцией, то выдается пустое значение.

Сообщение об ошибке, выдаваемое при отрицательном значении аргумента, порождается функцией:

```
overfact[]<< funcdef x {
  "Ошибка! Отрицательный аргумент":[(x,0):<] >>return;
}
```

В случае переполнения значащий результат выдает другая перегруженная функция, сигнализирующая об ошибке переполнения:

```
overfact[]<< funcdef x {
  "Ошибка! Слишком большой аргумент":[(x,n):>] >>return;
}
```

При аргументе, равном 0 или 1, вычисления осуществляются одной из следующих функций:

```
overfact[]<< funcdef x { 1:[(x,0):=] >>return; }
overfact[]<< funcdef x { 1:[(x,1):=] >>return; }
```

И, наконец, при попадании аргумента в диапазон от двух до максимально допустимого числа получаемый результат возвращается следующей функцией:

```
overfact[]<< funcdef x {
  ({(x, (x,1):-:overfact):*})
    :[((x,1):>, (x,n):<=):*]:[]:. >> return
}
```

Вызов функций с одинаковой сигнатурой осуществляется по общему имени в функции, осуществляющей синхронизацию альтернативных значений в списке данных, что ведет к избавлению от пустых элементов:

```
fact << funcdef x { (x:overfact):[] >> return }
```

Использование функций с одинаковой сигнатурой обеспечивает их эволюционное наращивание и безболезненное изменение в процессе инкрементальной разработки программы.

Эволюционное расширение при обработке динамических структур данных

Наряду с использованием предопределенных значений возможна обработка данных произвольной структуры. Текущая версия языка «Пифагор» поддерживает только динамическую типизацию. Формирование новых типов данных осуществляется неявно в ходе выполнения программы. Подобный механизм использовался в ранних языках функционального программирования, например LISP [5].

В Пифагоре, как и других языках с динамической типизацией, все предопределенные данные имеют признак (тег), задающий тип. Значение размещается непосредственно за тегом или доступно через указатель на некоторую область памяти. Любая операция перед выполнением анализирует теги аргументов и в соответствии с этим интерпретирует значение. Формально объект данных можно представить в виде двойки:

Структура элемента = <тип, величина>.

Наряду с неявной обработкой данных допускается выделять тип любого элемента. Для этого используется предопределенная операция **type**. Формируемая при этом величина принадлежит к «типовым» и имеет точно такую же организацию, как и любой другой аргумент. Ее специфика проявляется лишь в том, что типом аргумента является **type**.

Структура типового элемента = <type, значение типа>.

В качестве значения выступает один из предопределенных типов, к которым в языке относятся: целочисленный тип **int**; действительный тип **real**; символьный тип **char**; булевский тип **bool**; ошибочный тип **error** и другие. Следует отметить, что применение операции **type** к «типовому» элементу невозможно и ведет к ошибке интерпретации **TYPEERROR**, например:

int:type ⇒ TYPEERROR.

Типовые элементы могут участвовать в проверке на равенство и неравенство, что позволяет заранее выявить возможность выполнения последующих операций обработки данных.

Однако использование только базовых типов затрудняет написание больших программ и не позволяет явно фиксировать более мощные абстракции, переносимые между различными программами. Выявление подтипов, состоящих из композиций, построенных на основе базовых элементов, ведет к необходимости постоянного использования специально написанных функций проверки принадлежности заданной конфигурации. Для фиксации результатов проверки и исключения повторных вызовов этих же функций над уже проверенными данными нужно использовать явно задаваемые признаки, например числовые константы. Подобный прием затрудняет переносимость, так как может вести к неоднозначному толкованию признаков при объединении функций, написанных для разных программ.

Использование неявной типизации можно рассмотреть на примере манипуляции с набором условных геометрических фигур. Предположим, что существует список фигур, который содержит данные о треугольниках и кругах (их номенклатура в дальнейшем может расширяться). Необходимо разработать функцию, осуществляющую вычисление периметра для каждой из фигур, которые произвольно расположены в едином динамическом списке. Треугольник можно задать тройкой целых чисел, указывающих длины сторон, для круга достаточно одного числа, определяющего радиус.

Добавление новых фигур, проводимое при расширении программы, может привести к тому, что одинаковые структуры данных будут использоваться для задания различных понятий. Поэтому необходим дополнительный признак, обеспечивающий однозначную идентификацию. В соответствии с изложенным фрагмент программы, задающий описание списка геометрических фигур на языке программирования «Пифагор», может выглядеть следующим образом:

```
Triangle << const 1; // признак треугольника
```

```
Circle   << const 2; // признак круга
// Список из пяти фигур
Figures << const ((Triangle, (3,4,5)),(Circle, 10),
  (Circle, 7), (Circle, 1), (Triangle, (13,14,15)));
```

При вычислении периметра необходимо использовать полный разбор элементов списка, явно отделять признаки от значений и осуществлять все необходимые проверки на низком уровне абстракции. Например, определение периметра обобщенной геометрической фигуры будет состоять из набора операций анализа обрабатываемых данных.

```
// Нахождение периметра обобщенной фигуры
pi << const 3.1415;
fig_perimeter << funcdef figure {
  tag << figure:1; // выделение признака фигуры
  fig << figure:2; // выделение параметров фигуры
  // Использование признака для выбора формулы
  // и активизации вычислений
  tag^(
         // суммирование сторон треугольника
         {((fig:1,fig:2):+,fig:3):+},
         // формула для периметра круга
         {((2,pi):*,fig):*}
      ):. >>return;
};
// Получение списка периметров по списку фигур
all_perimeter << funcdef fig_list {
  // Вычисление одновременно для всех элементов списка
  (fig_list:[]:fig_perimeter) >>return;
};
```

Однако представленная функция осуществляет централизованное вычисление периметров отдельных фигур и не допускает эволюционного расширения. Для решения этой задачи достаточно использовать перегрузку и распределить вычисление периметров по отдельным специализациям перегруженной функции:

```
 // Нахождение периметра треугольника
fig_perimeter[] << funcdef figure {
  tag << figure:1; // выделение признака фигуры
  fig << figure:2; // выделение параметров фигуры
  // Использование признака для выбора формулы
  // и активизации вычислений
  [(tag,1):=]^(
         // суммирование сторон треугольника
         {((fig:1,fig:2):+,fig:3):+}
      ):[]:.
  >> return;
};

// Нахождение периметра окружности
fig_perimeter[] << funcdef figure {
  tag << figure:1; // выделение признака фигуры
  fig << figure:2; // выделение параметров фигуры
  // Использование признака для выбора формулы
  // и активизации вычислений
  [(tag,2):=]^(
```

```
            // формула для периметра круга
            {((2,pi):*,fig):*}
        ):[]:. >>return;
};
```

Использование обычной типизации при обработке сложных структур данных имеет следующий недостаток: признаки данных являются обычными целыми числами, поэтому легко перепутать не только отдельные фигуры, но и списки, несущие различную семантическую нагрузку. Кроме того, отсутствие составных типов, контролируемых встроенными средствами, все равно заставляет моделировать их, что ведет к значительному увеличению размеров программы. Для рассматриваемых геометрических фигур необходимо проверить принадлежность к заданной структуре и соответствие типов элементов требуемым значениям. В языках, использующих абстрактные типы данных, подобные манипуляции осуществляются гораздо эффективнее.

Применение пользовательских типов данных

Для преодоления рассмотренных проблем предлагается инструментальная поддержка механизма строгой динамической типизации на основе пользовательских типов. В язык вводится ряд дополнительных конструкций и понятий:

- определение пользовательского типа;
- сравнение пользовательских типов на равенство и неравенство;
- проверка на принадлежность некоторого значения величине, допустимой для заданного пользовательского типа;
- преобразование в пользовательский тип;
- разыменование пользовательского типа.

Определение пользовательского типа задается соответствующим предикатом, сопоставляющим проверяемый элемент с некоторым выражением. Если результат проверки является истиной, то элемент принадлежит проверяемому типу. Предикат оформляется в виде специальной функции **typedef**, возвращающей булевское значение. Ее обозначение регистрируется в таблице пользовательских типов. В качестве примера можно рассмотреть, как задаются треугольник и круг:

```
Triangle << typedef X {
  // Аргумент - список из трех целочисленных элементов
  [(((X:type,datalist):=,(X:|,3):=):*:int,1):+]^
  (
    false,
    {([(X:1:type,int),
       (X:2:type,int),
       (X:3:type,int)]:=):*}
  ):.  >> return
};
Circle << typedef X {
  // Аргумент - целочисленный атом
  (X:type,int):= >> return;
};
```

Сравнение пользовательских типов осуществляется точно так же, как и сравнение базовых типов языка: выделяется тип элемента функцией **type**, проверяется совпадение имен выделенного и проверяемого типа. Результат сравнения является истиной при совпадении имен типов. Ниже приводится пример использования сравнения пользовательских типов для обобщенной геометрической фигуры.

```
Figure << typedef X {
  // Аргумент - треугольник или круг
```

```
  X:type >> t
  ([(t, Triangle), (t, Circle)]:=):+ >> return;
};
```

Проверка на принадлежность позволяет выяснить возможность соответствия между динамически формируемыми данными и **typedef**. Для этого используется функция **in**. Она возвращает значение, которое получено в ходе выполнения предиката, заданного в описании пользовательского типа. Принадлежность позволяет в дальнейшем осуществить преобразование проверяемого аргумента в элемент пользовательского типа. Ниже представлены примеры использования функции принадлежности:

((10,20,15),Triangle):in ⇒ true

((10,20,15),Circle):in ⇒ false

(10,Circle):in ⇒ true.

Преобразование в пользовательский тип используется для формирования требуемых абстракций по принципу «обертки» преобразуемых данных и является расширением операции преобразования базовых типов. Суть заключается в получении нового значения элемента следующей структуры:

Элемент пользовательского типа =

<пользовательский тип, преобразуемый элемент>.

Само преобразование задается указанием пользовательского типа в качестве функции и осуществляется в зависимости от значения аргумента:

- если тип аргумента совпадает с типом в операции преобразования, то возвращается значение исходного аргумента;
- преобразование осуществляется только в том случае, если проверка аргумента на принадлежность функцией **in**, осуществляемая неявно, дает «истину»;
- во всех остальных случаях функция преобразования в пользовательский тип возвращает ошибку **TYPEERROR**.

Использование данной операции позволяет формировать необходимые абстракции при выполнении программы:

(10,20,15):Triangle ⇒ Треугольник со сторонами (10,20,15).

Описанная операция не обеспечивает автоматического преобразования пользовательских типов, даже если их значения принадлежать единому подмножеству. Это ограничение введено для более строгого контроля. Зачастую подобные преобразования бывают необходимы. В этом случае можно воспользоваться **разыменованием пользовательского типа**, заключающимся в выделении «обернутого» значения функцией **value**. Данная функция «отбрасывает» пользовательский тип, тем самым «обезличивая» преобразуемый элемент:

(10,20,15):Triangle:value ⇒ (10,20,15);

(10,20,15):Triangle:value:1:Circle ⇒ Круг радиусом 10.

Следует отметить, что попытка применить операцию разыменования к базовым типам ведет к генерации ошибки **VALUEERROR**:

10:value ⇒ VALUEERROR.

Приведенная выше функция вычисления периметров геометрических фигур может быть реализована следующим образом:

```
// Список из пяти фигур
Figures << const ((3,4,5): Triangle, 10:Circle,
7:Circle, 1:Circle, (13,14,15): Triangle);
// Нахождение периметра обобщенной фигуры
pi << 3.1415;
fig_perimeter << funcdef figure {
  fig << figure:value; // выделение параметров фигуры
```

```
    // Формирование селектора по типу фигуры
    tag << ([(figure:type,Triangle),
             (figure:type,Circle)]:=):?;
    // Использование признака для выбора формулы
    tag^(
          // суммирование сторон треугольника
          {((fig:1,fig:2):+,fig:3):+},
          // формула для периметра круга
          {((2,pi):*,fig):*}
        ):. >>return;
};
// Получение списка периметров по списку фигур
all_perimeter << funcdef fig_list {
  // Вычисление одновременно для всех элементов списка
  (fig_list:[]:fig_perimeter) >>return;
};
```

Сочетание пользовательских типов и перегрузки функций с одинаковой сигнатурой

Предлагаемые механизмы обеспечивают инструментальную поддержку для гибкой разработки функциональных параллельных программ. Возможно не только добавление новых обработчиков специализаций, но и изменение свойств ранее разработанных типов без модификации уже написанного кода.

Для построения эволюционно расширяемого обобщения достаточно воспользоваться промежуточной перегружаемой функцией, обеспечивающей проверку на используемый тип данных. Вызов этой функции внутри пользовательского определения обобщенного типа позволяет одновременно запустить все доступные проверки. Например, для организации эволюционно расширяемого типа обобщенной фигуры вводится перегружаемая функция **IsFigure**. Она проверяет исходные данные на треугольник и круг:

```
// Проверка на наличие треугольника
IsFigure[] << funcdef figure {
  figure:type >> t; (t, Triangle):= >>return;
};
// Проверка на наличие круга
IsFigure[] << funcdef figure {
  figure:type >> t; (t, Circle):= >>return;
};
```

Тип обобщенной геометрической фигуры осуществляет дизъюнкцию результатов проверки перегружаемой функции **IsFigure**:

```
// Описание фигуры, определяемое результатом
// выполнения перегруженной функции IsFigure
Figure << typedef X { (X:IsFigure):+ >> return; };
```

Добавление новой геометрической фигуры, например прямоугольника, происходит за счет описания его типа и добавления дополнительной специализированной проверки в виде очередной перегрузки функции **IsFigure**:

```
// Описание пользовательского типа, задающего
// прямоугольник как двухэлементный список
Rectangle << typedef X {
  [(((X:type,datalist):=,(X:|,2):=):*:int,1):+]^
  ( false,
    {([(X:1:type,int),(X:2:type,int)]:=):*}
```

```
  ):.  >>return
}
// Проверка на наличие прямоугольника
IsFigure[] << funcdef figure {
  figure:type >> t; (t, Rectangle):= >>return;
}
```

При наличии уже перегруженных функций добавление обработчика для фигуры, задающей новую специализацию, не представляет каких-либо проблем. Это обеспечивается независимым описанием используемых специализаций и отсутствием прямой связи между обобщенными объектами и обработчиками специализаций. Пусть первоначальное вычисление периметра будет разработано для треугольника и круга:

```
// Вычисление периметра треугольника
OverPerimeter[]<<funcdef figure {
   fig << figure:value;
   [(figure:type,Triangle):=]^
     {((fig:1,fig:2):+,fig:3):+} >>return;
}
// Вычисление периметра круга
OverPerimeter[]<<funcdef figure {
   fig << figure:value;
   [(figure:type,Circle):=]^
     {((2,3.14):*,fig):*} >>return;
}
```

Вычисление периметра прямоугольника осуществляется добавлением соответствующей перегруженной функции:

```
// Вычисление периметра прямоугольника
OverPerimeter[]<<funcdef figure {
   fig << figure:value;
   [(figure:type,Rectangle):=]^
     {((fig:1,fig:2):+,2):*} >>return;
}
```

Аналогичным образом могут быть реализованы и мультиметоды [6]. В этом случае каждый обработчик специализации, определяемый перегружаемой функцией, осуществляет проверку одной из комбинаций обрабатываемых специализаций.

Разработанный механизм эволюционного расширения функционально-параллельных программ ведет к гибкому добавлению новых функций в уже существующий код программы. Совместное использование динамической пользовательской типизации и перегрузки функций обеспечивает построение абстракций, используемых в эволюционно расширяемых параллельных программах.

СПИСОК ЛИТЕРАТУРЫ

1. Легалов, А. И. На пути к переносимым параллельным программам / А. И. Легалов, Д. А. Кузьмин, Ф. А. Казаков, Д. В. Привалихин // Открытые системы. – 2003. – № 5. – С. 36-42.
2. Легалов А. И. Инструментальная поддержка процесса разработки эволюционно расширяемых параллельных программ / А. И. Легалов // Проблемы информатизации региона. ПИР-2003: Материалы 8-й Всероссийской научно-практической конференции. – Красноярск, 2003. – С. 132-136.
3. Бен-Ари М. Языки программирования. Практический сравнительный анализ: Пер. с англ. / М. Бен-Ари. – М.: Мир, 2000. – 366 с.

4. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.
5. Маурер У. Введение в программирование на языке ЛИСП / У. Маурер. – М.: Мир, 1976. – 104 с.
6. Легалов, А. И. Мультиметоды и парадигмы / А. И. Легалов // Открытые системы. – 2002. – № 5. – С. 33-37.

EVOLUTION EXTENSION OF PROGRAMS IN FUNCTINAL PARALLEL PROGRAMMING LANGUAGE

A.I. Legalov, D.V. Privalihin

Program objects of «Pifagor» functional parallel language for evolution extension of program investigated. Overloaded functions and dynamic user types are used for it.