3. Использование базовых понятий функционального языка параллельного программирования

Особенности модели вычислений и синтаксиса ФЯПП накладывают отпечаток на методы и стиль программирования. Отсутствие операторов цикла позволяет писать потоковые программы без синхронизации перед входом в циклический фрагмент, но в то же время приводит к необходимости использования рекурсии. Рекурсии можно избежать, если алгоритм программируемой задачи предусматривает выполнение одной функции или списка функций над списком независимых аргументов. Тогда можно воспользоваться механизмом параллельных списков.

3.1. Применение параллельных списков

Использование механизма параллельных списков позволяет задавать массовый параллелизм вычислений, описывающий одновременную обработку нескольких потоков данных одной функцией.

Параллельный список аргументов

Рассмотрим пример умножения элементов числового вектора на скаляр. В качестве аргумента предполагается передавать двухэлементный список, первый элемент которого является вектором, а второй – скаляром:

$$((x_1, x_2, \ldots, x_n), y)$$

где $x_1, x_2, \ldots x_n$ – элементы вектора, y – скаляр.

```
// Функция, возвращающая произведение вектора на скаляр
VecScalMult << funcdef Param
// Формат аргумента: ((x1, x2, … xn), y),
// где ((x1, x2, … xn)- числовой вектор, y – числовой скаляр
{
      // Выделение из списка параметров первого элемента –
      // вектора и обозначение его идентификатором X
      X << Param:1;
      // Выделение из списка параметров второго элемента –
      // скаляра и обозначение его идентификатором y
      y << Param:2;
      // Определение длины вектора и обозначение полученного
      // значения идентификатором Len
      Len << X:|;
      // Формирование вектора длины Len путем дублирования скаляра
      // y Len раз и обозначение его идентификатором V
      V << (y,Len):dup;
      // Объединение двух векторов X и Y в двухстрочную матрицу
      ((X,V)
      // Транспонирование полученной матрицы
      // в список двухэлементных подсписков
      :#
      // Преобразование полученного списка в параллельный
      :[]
      // Параллельное выполнение операции умножения над всеми
      // парами, охват полученного параллельного списка круглыми
      //скобками и выход из функции в возвратом полученного списка
      :*)>>return
}
```

Пример выполнения:

```
((3,5.02,-2,0,1.5),10): VecScalMult
⇒ (30,5.020000e+001,-20,0,1.500000e+001)
```

Комментарии, используемые в данном случае для пояснения особенностей кода, явно избыточны для такой программы и во многом затрудняют восприятие кода. Поэтому ниже представлен исходный текст этой же функции, но без лишних комментариев.

```
// Функция, возвращающая произведение вектора на скаляр
VecScalMultNoComment << funcdef Param
// Формат аргумента: ((x1, x2, … xn), y),
// где ((x1, x2, … xn)- числовой вектор, y – числовой скаляр
{
     X << Param:1;
     y << Param:2;
     Len << X:|;
     V << (y,Len):dup;
     ((X,V):#:[]:*) >>return
}
```

Кроме этого, функциональный стиль, поддерживаемый языком, позволяет, во многих случаях сводить исходный текст функции к одному оператору, определяющему все вычисления. Подобная версия программы для рассматриваемого примера будем выглядеть следующим образом:

```
// Функция, возвращающая произведение вектора на скаляр,
// написанная с использованием минимального числа операторов
VecScalMultBrief << funcdef Param
// Формат аргумента: ((x1, x2, : xn), y),
// где ((x1, x2, : xn)- числовой вектор, y - числовой скаляр
{
    ((Param:1,(Param:2,Param:1:|):dup):#:[]:*) >>return
}
```

Следует отметить, что не всегда целесообразно придерживаться подобного стиля, так как затрудняется восприятие программы и происходит дублирование одних и тех же вычислений.

Параллельный список функций

Функции тоже могут задаваться параллельным списком, определяя тем самым множество потоков функций над одним потоком данных. Следующий пример иллюстрирует параллельное нахождение суммы, разности, произведения и частного двух чисел:

```
// Функция, параллельно находящая сумму, разность,
// произведение и частное двух чисел - элементов
// двухэлементного списка аргументов
ParAddSubMultDiv << funcdef Param
    // Формат аргумента: (число, число),
{
     // Осуществляем параллельное нахождение суммы,
     // разности, произведения и частного двух чисел
     // с возвращением списка результатов
     (Param:[+,-,*,/]) >>return
}
```

Пример выполнения:

```
(3,5): ParAddSubMultDiv ⇒ (8,-2,15,6.000000e-001)
```

3.2. Использование задержанных списков

Для программирования вычислительных алгоритмов, предусматривающих ветвление, применяются задержанные списки с последующим выбором и раскрытием элемента, соответствующего дальнейшим вычислениям. Рассмотрим пример функции, находящей абсолютное значение скалярного аргумента:

```
// Функция, возвращающая абсолютное значение аргумента
Abs << funcdef Param
  // Аргумент является числом
{
      // Задержанное выражение, результат которого –
      // аргумент с изменённым знаком
      ({Param:-},
      // Второй аргумент не нуждается в задержке,
      // поскольку не влечёт никаких вычислений
      Param):
      // Параллельное сравнение аргумента функции с нулём на
      // «меньше» и «больше либо равно» с последующим охватом
      // результата круглыми скобками
      [(Param,0):(<,=>)
      // Преобразование списка булевских скалярных величин в
      // список целочисленных констант, значения которых
      // соответствуют позициям булевских величин, имеющих
      // значение «истина». Поскольку условия «меньше» и «больше
      // либо равно» исключают друг друга, то результатом данной
      // операции будет целочисленная константа, имеющая
      // значение, равное 1 или 2, соответствующее первому или
      // второму элементу списка задержанных вычислений.
      // Применение этой константы к списку задержанных вычислений
      // повлечёт за собой выбор соответствующего элемента, но
      // раскрытия задержки не произойдёт, пока не будет предпринята
      // попытка интерпретации задержанного элемента
      :?]
      // Раскрытие задержки и завершение функции с возвратом
      // результата
      :. >>return
}
```

Без комментариев текст программы выглядит следующим образом:

```
// Функция, возвращающая абсолютное значение аргумента
Abs << funcdef Param
  // Аргумент является числом
{
  ({Param:-}, Param): [(Param,0):(<,=>) :?]:. >>return
}
```

Примеры выполнения:

```
-3: Abs ⇒ 3
5: Abs ⇒ 5
```

```
0: Abs ⇒ 0
```

3.3. Использование параллельной рекурсии

Если вычислительный алгоритм предусматривает древовидное или рекуррентное использование функции для множества однотипных аргументов, количество которых может быть произвольным (например, функция суммирования всех элементов вектора), то в этом случае применяется рекурсивная декомпозиция списка аргументов, на самом нижнем уровне которой выполняется операция над одноэлементными или двухэлементными списками, полученными в результате разложения. После этого следует обратный ход со сверткой отдельных результатов. Рассмотрим вычисление суммы элементов числового вектора произвольной длины.

```
// Функция, возвращающая сумму элементов вектора
VecSum << funcdef Param
     // Формат аргумента: (x1, x2, … , xn)
     // где x1, x2, … , xn - числа
{
  // Обозначим длину вектора идентификатором Len
  Len<<Param:|;
  // Параллельно сравниваем длину списка с числовой
  // константой 2 на «меньше», «равно» и «больше», преобразуем
  // список булевских атомов в список числовых атомов, который
  // затем применяем в качестве функции к списку задержанных
  // выражений. Поскольку знаки логических операций
  // взаимоисключающи, то результатом будет числовая константа,
  // значение которой соответствует номеру элемента списка
  // задержанных выражений
  return<< .^[((Len,2):[<,=,>]):?]^
  (
  // Первый элемент списка задержанных выражений: при длине
  // аргумента меньшей двух преобразуем одноэлементный список
  // в параллельный, что приводит к выделению его единственного
  // элемента
     {Param:[]},
     // Аргумент имеет длину 2, поэтому просто суммируем
     // его элементы
     {Param:+},
     // Третий элемент списка задержанных вычислений реализован
     // в виде блока, поскольку для удобочитаемости и наглядности
     // записи данного задержанного выражения были использованы
     // идентификаторы.
     {
       block {
          // Идентификатором OddVec обозначен список данных,
          // составленный из нечётных элементов аргумента
          OddVec << Param:[(1,Len,2):..];
          // Идентификатором EvenVec обозначен список данных,
          // составленный из чётных элементов аргумента
          EvenVec << Param:[(2,Len,2):..];
          // Оба списка объединяются в параллельный список над
          // которым выполняется функция суммирования элементов
          // списка, что приводит к параллельному рекурсивному
```

```
            // выполнению этой функции
            ([OddVec,EvenVec]: VecSum
            // Результаты параллельного выполнения двух
            // экземпляров функции VecSum суммируются
            ):+
            // Осуществляется выдача результата из блока
            >>break
        } // конец блока
      } // конец задержанного списка
  )
}
```

Без избыточных комментариев функция будет выглядеть следующим образом:

```
// Функция, возвращающая сумму элементов вектора
VecSum << funcdef Param
     // Формат аргумента: (x1, x2, … , xn)
     // где x1, x2, … , xn - числа
{
  Len<<Param:|;
  return<< .^[((Len,2):[<,=,>]):?]^
  (
     {Param:[]},
     {Param:+},
     {
       block {
          OddVec << Param:[(1,Len,2):..];
          EvenVec << Param:[(2,Len,2):..];
          ([OddVec,EvenVec]: VecSum):+
          >>break
       } // конец блока
     } // конец задержанного списка
  )
}
```

Пример выполнения:

```
(-3, 6, 10, 25, 0): VecSum ⇒ 38
```

3.4. Использование функций в качестве параметров

Если в предыдущем примере нам понадобится находить не сумму, а произведение элементов списка произвольной длины, то необходимо переписывать всю программу. Однако ФЯПП позволяет написать общую функцию декомпозиции, в качестве первого аргумента которой будет выступать обрабатываемый вектор, а вторым аргументом может быть любая бинарная функция, которую предполагается выполнять при свёртке дерева, полученного при декомпозиции. Рассмотрим функцию, осуществляющую общую декомпозицию списка с последующим выполнением, при свёртке, функции, переданной в качестве второго параметра.

```
// Функция, осуществляющая декомпозицию списка с последующим
// вычислением функции, переданной в качестве второго параметра.
BinTreeReduction << funcdef Param
  // Формат аргумента: (x1, x2, … , xn)
  // где x1, x2, … , xn – элементы обрабатываемого вектора
  // f – обрабатывающая функция
{
```

```
  Len << Param:1:|; // длина списка-аргумента
  Func << Param:2;  // Переданная функция
  return<< .^[((Len,2):[<,=,>]):?]^
    (
      {Param:1:[]},   // Первый элемент при длине меньшей двух
      {Param:1:Func}, // Свертка с использованием параметра-функции
      {
        // Блок, определяющий рекурсивные вычисления
        block {
          // нечетные элементы
          OddVec  << Param:1:[(1,Len,2):..];
          // четные элементы
          EvenVec << Param:1:[(2,Len,2):..];
          // Рекурсивная параллельная декомпозиция со сверткой
          ([(OddVec,Func),(EvenVec,Func)]: BinTreeReduction):Func

        >>break} // конец блока
     } // конец третьего задержанного аргумента
  ) // конец всех альтернатив
}
```

Следует отметить, что использование функции в качестве параметра пока допустимо только внутри другой функции. Предполагается, что в дальнейших версиях интерпретирующей среды имена предопределенных и оттранслированных функций можно будет задавать в исходных данных, подгружаемых в программу. Различные варианты использования функции в качестве параметра приведены в следующей тестовой функции:

```
// Функция, тестирующая различные варианты
// параметров-функций в BinTreeReduction
BinTreeReductionTest << funcdef Param
  // Формат аргумента: (x1, x2, : , xn)
  // где x1, x2, : , xn - числа
{
  (
    (Param,+):BinTreeReduction,
    (Param,*):BinTreeReduction,
    (Param,Min):BinTreeReduction,
    (Param,AbsAdd):BinTreeReduction
  ) >>return
}
```

В представленном тесте функции “+” и “*” являются операциями, предопределенными в языке. Функции **Min** (находит минимум для двух элементов списка) и **AbsAdd** (суммирует абсолютные величины двух чисел) реализованы программно следующим образом:

```
// Выбор минимального значения для
// двухэлементного числового вектора
Min << funcdef Param
  // Формат аргумента: (число1, число2)
{
  Param:[Param:(<=,>):?] >>return
}

// Суммирование абсолютных значений
// двухэлементного числового вектора
```

```
AbsAdd << funcdef Param
  // Формат аргумента: (число1, число2)
{
  [Param:1,Param:2]:(Abs):+ >>return
}
```

Пример использования:

```
(-3, 6, -1, 2, -5): BinTreeReductionTest ⇒ (-1,-180,-5,17)
```