УДК 004.432.42

# СЕМАНТИКА ОПЕРАЦИЙ ФУНКЦИОНАЛЬНО-ПОТОКОВОГО ПАРАЛЛЕЛЬНОГО ЯЗЫКА «ПИФАГОР»

**Кропачева М.С.**
**Научный руководитель — д.т.н., профессор Легалов А.И.**
***Сибирский федеральный университет***

В настоящее время параллельные вычислительные системы (ПВС) широко используются для решения различных задач, что ведет к постоянному росту числа разрабатываемых прикладных параллельных программ. Однако параллелизм значительно усложняет проверку корректности (правильности) написанного кода, что связано с императивностью существующих систем программирования и их ориентацией на физические вычислительные ресурсы. Это ведет к появлению в программах не только логических ошибок, обуславливаемых некорректными вычислительными операциями, но и конфликтных ситуаций, связанных с неправильным взаимодействием ресурсов ПВС. В качестве альтернативы существующим системам программирования предлагается использование функционально-потоковой параллельной парадигмы, ориентированной на представление программы в виде информационного графа с управлением по готовности данных. Также предполагается, что программа выполняется в неограниченных ресурсах, что ведет к исключению конфликтов соответствующего типа. На сегодняшний день использование этой парадигмы не обеспечивает реализацию эффективных вычислений, но облегчает отладку параллельных программ и проверку их корректности, что позволяет в дальнейшем использовать полученный и отлаженный код для переноса в системы программирования, исполняемые в реальной среде. Это, в сочетании со средствами автоматизации программирования, позволяет улучшить надежность программ и повысить эффективность процесса их разработки.

Для поддержки функционально-потоковой парадигмы программирования предложен язык «Пифагор». К его основным характеристикам относится отсутствие привязки к ресурсам системы, а значит можно избежать конфликтов, связанных с совместным использованием памяти параллельными процессами, что, в свою очередь, упрощает анализ корректности программы. Однако на данный момент семантика языка до конца не формализована, поэтому цель работы заключается в формальном описании выполняемых операций, чтобы её можно было использовать для анализа корректности программ.

Программа на функциональном языке представляется в виде информационного графа (ациклический ориентированный граф). Вершины графа представляют программо-формирующие операторы, а дуги — пути передачи информации. Существуют следующие типы операторов: оператор интерпретации, константный оператор, оператор копирования данных, оператор группировки в список, оператор группировки в параллельный список и оператор группировки в список задержанных вычислений. Выполнение программы соответствует разметки дуг в информационном графе, при этом вычисления завершаются, когда все дуги оказываются размеченными. Правила распространения разметки по графу складываются из общих правил межоператорных переходов, правил срабатывания программо-формирующих операторов, правил выполнения оператора интерпретации, правил эквивалентных преобразований операторов и связей информационного графа. Две последние группы правил можно объединить при описании семантики языка.

Оператор интерпретации обеспечивает преобразование входного набора данных X, выступающего в качестве аргумента, в выходной набор Y, играющего роль результата,

используя при этом входной набор F в качестве функции, определяющей алгоритм преобразования. Аргумент и функция могут являться результатами предшествующих вычислений. Оператор интерпретации запускается по готовности данных, что фиксируется появлением разметки на входных дугах. Получение результата задается разметкой выходной дуги. При текстовом описании оператор интерпретации имеет две формы: постфиксную (X:F) и префиксную (F^X). В постфиксной форме функциональные преобразования аргумента записывается следующим образом: X:F→Y.

Все (унарные) функции F, поступающие оператору интерпретации можно разделить на два подмножества: множество предопределенных функций языка и множество функций (например, арифметические функции, функции сравнения и др.), порождаемых при программировании (пользовательские функции). Необходимо прописать семантику для предопределенных функций, а именно четко указать их область определения и область значения для всевозможных типов аргументов. Для правильных входных значений функция должна выдавать соответствующий результат и ошибку для всех остальных случаев. Так как пользовательские функции фактически являются комбинацией предопределенных функций, то, исходя из области определений входных аргументов, можно будет получить область значений при применении соответствующих семантических правил. Для описания семантики разобьём процесс выполнения оператора интерпретации на два этапа:

1) анализ и преобразование операндов, которое не зависит от операции;
2) выполнение операции.

Тогда получаем две группы правил, которые применяются последовательно и независимо друг от друга.

Первый этап заключается в преобразовании операндов по правилам эквивалентных преобразований. Правила приведены в таблице 1. Они применяются в указанном порядке, и, после применения одного из правил, следующее ищется с начала списка. Переход к этапу выполнения происходит только в том случае, если ни одно из правил нельзя применить.

**Таблица 1.** Первичный анализ с использованием правил эквивалентных преобразований.

**0.** 0.1. Анализ кратности, в результате которого параллельный список $[v_1, v_2, \ldots, v_n]$ представляется в виде $<n, <type_1, v_1>, \ldots, <type_n, v_n>>$, где n — кратность.
0.2. Эквивалентность элемента и параллельного списка кратности один
$x \equiv [x]$

**1.** Раскрытие задержанных списков
1.1. $\{X\} : F \rightarrow [X] : F$
1.2. $X : \{F\} \rightarrow X : [F]$
1.3. $\{X\}:\{F\} \rightarrow [X] : [F]$
где X и F произвольные значения

**2.** Раскрытие параллельных списков
2.1. $[x_1, x_2, \ldots, x_n] : F \rightarrow [x_1:F, x_2:F, \ldots x_n:F]$
2.2. $X : [f_1, f_2,..., f_n] \rightarrow [X:f_1, X:f_2, \ldots, X:f_n]$
2.3. $[x_1, x_2, \ldots, x_m ] : [f_1, f_2,..., f_n] \rightarrow$ **[** $[x_1:f_1, x_1:f_2, ..., x_1:f_n], [x_2:f_1, x_2:f_2, ..., x_2:f_n], ..., [x_m:f_1, x_m:f_2, ..., x_m:f_n]$ **]**

**3.** Преобразования списка данных
3.1. Слияние параллельных списков в списке данных
$(x_1, x_2, \ldots, x_{i-1}, [\mathbf{X_i}], x_{i+1}, \ldots, x_n) \rightarrow (x_1, x_2, \ldots, x_{i-1}, \mathbf{X_i}, x_{i+1}, \ldots, x_n)$

3.2. Удаление пустых элементов «•»

$(x_1, x_2, \ldots, x_{i-1}, \bullet, x_{i+1}, \ldots, x_n) \rightarrow (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$

3.3. X:(F) → (X:[F])

**4.** Преобразования асинхронных списков

4.1. Удаление пустых элементов «•»

$\text{asynch}(x_1, x_2, \ldots, x_{i-1}, \bullet, x_{i+1}, \ldots, x_n) \rightarrow \text{asynch}(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$

Этап выполнения операции напрямую зависит от самой операции. В самом простом случае он может описываться с помощью естественного языка. Например, функция выбора элемента из списка (целочисленная константа) в качестве аргумента принимает список любой размерности, содержащий элементы любого типа. Результат зависит от значения константы. Если она положительна и находится в диапазоне от единицы до величины равной длине списка, то результат — элемент с соответствующим порядковым номером. Если значение константы превышает длину списка, то выдается ошибка выхода за его границы (BOUNDERROR). В случае отрицательной константы происходит исключение элемента с соответствующим номером из списка или выдается ошибка, если абсолютное значение константы превышает длину списка. Нулевое значение константы возвращает пустое значение, обозначаемое «.».

В нашем случае требуется более формальное описание, которое гарантирует отсутствие неучтенных вариантов значений входных данных и может быть использовано при анализе корректности. Для этого, опишем семантику в следующем виде: характеризуем аргумент необходимыми условиями (например, «аргумент имеет целый тип» или «значение аргумента не равно нулю»), которые запишем в виде выражения. В зависимости от значения выражения можем либо перейти к проверке другого условия, либо выдать результат вычисления. Для каждого выражения должен рассматриваться случай – «все остальные значения этого выражения». Это позволит корректно обработать любой аргумент, а при расширении семантики функции легко добавить новые условия, отделив их от множества «остальных значений».

Выражение будем записывать на языке «Пифагор», используя три основные функции: определение типа аргумента (p:type), получение i-го элемента списка (p:i) и определение длины списка (p:|). Выражения записываются через эти функции и функции, которые определены раньше этого выражения. А также при необходимости вставляем комментарии на естественном языке. Значения выражения — те значения, которые может принять выражение, *else* — все остальные значения. Для описания результата используем обычные логические (∧, ∨, ¬), арифметические операции (+, -, *, /) и знаки сравнения ( =, !=, >, <, >=, <=) или естественный язык.

В качестве примера в таблице 2 приведем операцию выбора элемента из списка, описанную выше. Считаем, что операции сравнения и проверки на равенство уже были определены в предыдущих правилах.

**Таблица 2**. Семантика функции выбора элемента из списка (**p:<int,b>**).

| **№** | **Выражение** | **Значения выражения** | **Результат/Действия** |
|---|---|---|---|
| **1** | p:type | datalist | → 2 (переходим к пункту 2) |
| | | *else* | **BASEFUNCERROR** |
| **2** | (b,0):= | true | **.** (пустое значение) |
| | | *else* | → 3 |

| | | | |
|---|---|---|---|
| **3** | абсолютное значение b не превышает длину списка p | да и b>0 | **$p_b$**, где p=($p_1$,..., $p_n$), n>=b |
| | | или значение выражения (true, true, false, false) | |
| | | да и b<0 | **($p_1$,..., $p_{b-1}$, $p_{b+1}$,..., $p_n$)**, где p=($p_1$,..., $p_n$), n>=b |
| | или выражение ( (p:\|,b):>=, (b,0):>, (p:\|,b:-):>=, (b,0):<) | или значение выражения (false, false, true, true) | |
| | | *else* | **BOUNDERROR** |

Проверку корректности программы можно осуществлять, анализируя разметку программы для абстрактных данных, то есть, не конкретных значений, как в тестировании, а данных, описанных в общем виде некоторыми правилами. Например, аргумент X имеет целочисленный тип и значения от нуля до максимального целого. Совокупность условий на начальные (входные) данные программы формирует предусловие. Результат работы программы тоже можно описать с помощью условий для выходного набора значений – это будет постусловием. Тогда, используя формализованную семантику языка, можно (с помощью формальных методов) вывести постусловие для конкретного оператора из имеющегося предусловия. Выполнив эти действия для всей программы, получаем постусловие результата вычислений для имеющегося предусловия. Это постусловие можно сравнить с тем постусловием, которое должно было быть у корректной программы. Результат такого сравнения и позволяет судить о правильности программы. Вывод: описание семантики позволит анализировать корректность программы посредством вывода постусловия из предусловия.