Система типов языка функционально-потокового параллельного программирования*

И.В. Матковский

Сибирский Федеральный Университет

В работе предлагается расширение системы типов языка функционально-потокового параллельного программирования «Пифагор» на основе проведенного анализа существующих подходов. Рассматривается возможности используемых в нем методов типизации данных и пути дальнейшего развития. Описывается система статической типизации, обеспечивающая дополнительный анализ корректности данных во время компиляции. Предлагается метод трансформации типов данных во время выполнения программ, комбинирующий статическую и динамическую типизацию.

Ключевые слова: системы типов, функционально-потоковое параллельное программирование, типизация, статическая типизация, динамическая типизация, проверка типов.

1. Введение

При разработке современных языков программирования большое внимание уделяется надежности создаваемого кода. Особенно это важно для языков, предназначенных для написания прикладного программного обеспечения, где не нужно управлять вычислительными ресурсами, а основной задачей является обработка данных в соответствии с алгоритмами предметной области. Одним из ключевых факторов при разработке таких языков является поддержка однозначности выполняемых преобразований данных как на уровне базовых операций и операторов языка, так и на уровне более сложных конструкций, создаваемых на их основе: процедур, функций, классов и других. Для этого практически все языки программирования включают системы типов, обеспечивающие однозначное толкование обрабатываемых данных и производимых над ними операций. При этом для достижения однозначности используются статическая и динамическая типизация, а также их сочетания, обеспечивающие специфические характеристики различных языков программирования.

Можно выделить три основных уровня типизации, обеспечивающие альтернативные варианты однозначного формирования кода (однозначности):

- образная интерпретация типов данных;
- статическая типизация;
- динамическая типизация.

На уровне машинных команд современных архитектур данные рассматриваются только как последовательности битов в памяти и, с точки зрения программного кода, типов не имеют. В таких случаях можно говорить об образной интерпретации типов данных. Команды работают с обезличенными машинными словами, и контроль корректности получаемых данных производится вручную. В этой ситуации однозначность кода обеспечивается выполняемыми командами, трактующими области памяти данных как операнды конкретного типа.

Статическая типизация предполагает, что тип данных однозначно определяется при описании переменных. Этот тип ограничивает возможность использования данных на заданном множестве операций, что позволяет контролировать корректность программы и выявлять ошибки, связанные с неправильным использованием данных уже на этапе компиляции. С другой стороны, написание программ таким образом требует дополнительных усилий со стороны программиста. Не всегда конкретный тип данных известен заранее. Поэтому достаточно часто приходится прибегать к его анализу программными средствами, используя явные проверки или

* Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта 17-07-00288.

прибегая к методам динамического полиморфизма, реализуемого, например, с применением наследования и виртуализации.

Динамическая типизация предполагает, что связывание фрагментов данных с типами происходит в ходе вычислений после получения конкретных значений в качестве результатов выполнения различных вычислений. Однозначность выполнения операций при этом обеспечивается на этапе выполнения программы после того, как операция, полиморфная относительно своих аргументов, определит их тип, произведет соответствующие вычисления и сформирует результат. Тип результата в таком случае определяется из семантики данной операции. Это позволяет разрабатывать код, менее зависимый от типов входных данных. Повышение надежности, с другой стороны, требует вводить дополнительные проверки типов данных в код программы или рисковать ростом количества трудно отслеживаемых логических ошибок.

Наряду с динамической и статической типизацией используется понятие сильной и слабой типизации. Определения этих терминов несколько расплывчаты, однако во многих случаях систему типизации можно охарактеризовать достаточно четко. Среди отличительных черт сильной типизации выделяют запрет на неявные приведения типов и тщательное отслеживание корректности типов данных операциями на этапе выполнения программы. Сильная статическая типизация часто используется в языках функционального программирования, среди которых можно отметить SISAL [1], Scala [2], Haskell [3]. Это обуславливается необходимостью эффективного контроля данных во время компиляции, позволяющим не проводить дополнительные проверки типов во время выполнения, тем самым обеспечивая дополнительное повышение производительности. Вместе с тем, в ряде функциональных языков, например Erlang [4] и Lisp [5], применяется сильная динамическая типизация.

Язык Пифагор [6] разработан в рамках функционально-потоковой парадигмы параллельного программирования и предназначен для архитектурно-независимого параллельного программирования. В языке в настоящий момент в качестве основной используются традиционная сильная динамическая типизация [7], а также динамическая типизация, определяемая пользователем, позволяющая описывать типы данных как функции, обеспечивающие проверки на эквивалентность и преобразования в ходе вычислений [7]. Проверка принадлежности отдельных фрагментов данных к тем или иным типам производится только на этапе выполнения программы. В соответствии с семантикой языка возможно прямое преобразование базовых типов друг в друга, а также базовых типов в типы данных, определяемых пользователем.

Вместе с тем, дальнейшее развитие языка во многом связано с использованием для повышения надежности разрабатываемых программ методов формальной верификации, обеспечивающих доказательство корректности создаваемых функций [8]. Эффективность этих методов во многом зависит от наличия в языке статической типизации, которая, помимо этого, позволяет порождать более эффективное выходное представление. Также возможно информацию о типах можно использовать и при оптимизации программы [9].

2. Расширение системы типов в языке “Пифагор”

Добавление в уже существующий язык новых конструкций должно согласовываться с ранее реализованными синтаксисом и семантикой. Ниже рассмотрен ряд особенностей существующей системы типов и варианты ее расширения.

2.1 Описание пользовательских динамических типов

В основе описания пользовательских типов в языке функционально-потокового параллельного программирования “Пифагор” специализированные функции-предикаты typedef, проверяющие соответствие обрабатываемого значения определенному шаблону [7, 10]. При соответствии шаблону возвращается логическое значение “истина”. Подобное описание типа позволяет вводить достаточно сложные алгоритмические проверки, допускающие анализ как структуры получаемого значения, так и входящих в это значение величин различных полей. Данная реализация по логике близка контрактному программированию, представленному, в частности, в языке Eiffel [11]. Соответствие данному шаблону позволяет осуществлять во время выполне-

ния различные приведения типов, а также их сравнения без использования дополнительных проверок.

Для демонстрации особенностей типа, определяемого пользователем можно привести следующий пример, задающий описание прямоугольника (рис.1).

```
// Тип данных "прямоугольник"  – список данных из ровно
двух элементов,
// каждый из которых является целочисленным значением.

Rectangle << typedef X {

//первая проверка: является ли входной аргумент X спи-
ском данных
//вторая проверка: аргумент X состоит из двух элементов?
[(((X:type,datalist):=, (X:|,2):=):*:int,1):+]^
  (
//не пройдена первая или вторая проверка
    false,
//оба элемента являются целыми атомами?
    {([(X:1:type,int),
       (X:2:type,int)]:=):*}
  ):.  >> return
};
```

Рис. 1. Описание типа "прямоугольник"

На основе прямоугольника можно построить квадрат, который содержит описания двух сторон равной длины. Первая проверка определяет, является ли значение прямоугольником. Вторая проверка предназначена для анализа равенства сторон (рис.2).

```
//Тип данных "квадрат"  – прямоугольник с равными сторо-
нами

Square << typedef X {
// является ли входной аргумент X прямоугольником
 [(X:type,Rectangle):=:int]^
  (
    false,              //не пройдена первая проверка
    {([(X:1,X:2):=}     //стороны равны?
  ):.  >> return
};
```

Рис. 2. Описание типа "квадрат"

Ввиду динамической природы данного метода типизации, она может осуществляться только во время выполнения программы. Однако в ряде случаев более естественным и эффективным было бы использование статической типизации, что позволило бы не только обеспечивать генерацию более эффективного выходного представления, но и реализовать более глубокие методы оптимизации кода.

2.2 Особенности преобразования пользовательских типов

Особенности, присущие динамической типизации ведут к представлению данных в виде двоек: <type, value>, где type – тег типа данных, value – непосредственно данные. Например, целочисленное значение 10 задается в виде "<int,10>", а список из двух целочисленных значений (10,20) – в виде "<datalist, (<int,10>,<int,10>)>" Такой список соответствует правилам из описания типа Rectangle, однако он не будет считаться значением такого типа данных до явного приведения.

Явное преобразование типов можно продемонстрировать следующим примером:

```
Value1 << (10,10);
Value2 << Value1:Square;
Value3 << Value1:Rectangle;
Value3 << Value4:Square;
```

Рис. 3. Пример преобразования типов

Записанный в Value1 фрагмент данных будет иметь внутреннее представление следующего вида

Value1 == <datalist, (<int,10>,<int,10>)>

Так как тип Value1 не равен Rectangle, то попытка преобразовать его к типу Square приведет к ошибке типизации. В результате сформируется ошибка преобразования типа, которая и будет обозначена как Value2:

Value2 == <error, TypeError>

Попытка привести Value3 к типу Rectangle закончится успешно, так как тот задается именно как список данных из двух элементов. Таким образом, в Value3 будет эквивалентен следующему типу:

Value3 == <Rectangle, <datalist,(<int,10>,<int,10>)>>

Приведение Value4 к типу Square тоже закончится успешно. Описание типа Square задается на основе типа Rectangle:

Value4 == <Square,<Rectangle, <datalist,(<int,10>,<int,10>)>>>

Использование тегов позволяет ускорить процесс анализа типа данных в простых ситуация, когда требуется проверить только тип данных, без анализа структуры самих данных.

Недостатком типов, задаваемых пользователем, является цепочка, выстраивающаяся из типов данных, так как последовательные преобразования не стирают информацию о старых типах данных. С одной стороны это может быть полезным, так как позволяет отследить наследование формируемых значений. Однако в ряде случаев подобные построения являются ненужными и замедляют обработку данных. Поэтому, наряду с использованием вложенного подхода необходимо предусмотреть систему преобразования, позволяющую заменять одни пользовательские типы на другие, обеспечивая при необходимости не только замену тегов, но и преобразование значений.

2.3 Проверка типов

В существующей версии языка проверка на принадлежность значения типу можно осуществить следующими способами:

- тип значения можно получить с помощью оператора type с последующим равнением с заданным типом. Проверка будет успешной только для тех значений, которые имеют тип, эквивалентный проверяемому;
- значение приводится к нужному типу. Если приведение пройдет успешно, результатом станет новое значение, способное пройти проверку на сравнение типов. В противном случае выходное значение будет ошибкой TYPE_ERROR.

В обоих случаях проверка производится во время выполнения программы. Даже в том случае, если типы данных не нуждаются в динамической проверке и уже известны при написании программы, все равно проверки и преобразования будут проходить только во время выполнения. Вместе с тем, внедрение в язык статической типизации позволит в ряде случаев осуществлять требуемые проверки во время компиляции, что также позволит повысить эффективность выходного представления. Особенно это важно при анализе аргумента и значения функции, что позволит формировать однозначные интерфейсы, предоставляемые в библиотечном коде.

3. Расширение системы типов в языке “Пифагор”

Проведенный анализ показывает, что введение в язык статической типизации, методов приведения статических типов и методов преобразования типов данных позволяет повысить

гибкость языка, обеспечить дополнительные возможности по формальной верификации кода, и повысить эффективность формируемого выходного представления.

3.1 Включение статической типизации

Для включения в язык статической типизации необходимо учесть особенности существующего синтаксиса языка программирования «Пифагор». Особенностью статического описания типов является отсутствие алгоритмического анализа данных. Поэтому определение статического типа можно реализовать с использованием того же описателя typedef, что и при определении динамического пользовательского типа:

статический_тип = идентификатор "<< typedef " тип.
тип = “@” статический_пользовательский_тип | тип_атом | составной_тип.
тип_атом = "int" | "float" | "bool" | "char" | "spec" | "delay" | "error" | "extern"
составной_тип = ["datalist"] "(" [тип{“,” тип | “…” }] ")"[“.” размер] |
["parlist"] "["тип{“,”тип}"]" |
["asynclist"] "<(" тип{“,” тип} ")"

Использование таких определений в программе показано ниже (рис. 4).

```
//прямоугольник задается двумя целочисленными сторонами
Rectangle << typedef @(@int,@int)
//треугольник задается тремя целочисленными сторонами
Triangle << typedef @(@int,@int,@int)
//список, состоящий из одного или более значений типа
int
IntList << typedef @(@int,…)
// список, состоящий из значения типа bool и одного или
более значений типа int
TailIntList << typedef @(@bool, @int,…)
// список, состоящий из 7 элементов
SizedList << typedef @().7
// список, состоящий из 7 элементов типа int
SizedIntList << typedef @(@int,…).7
```

Рис. 4. Пример определений статических типов

В приведенном выше примере описано два типа данных – Rectangle (прямоугольник) и Triangle (треугольник). Прямоугольник задается двумя целочисленными сторонами, треугольник – тремя целочисленными сторонами.

Для обработки статических объявлений компилятором необходимо ввести следующие ограничения:

- статическое объявление может использовать только встроенные и статически объявленные типы данных; использование объявленных пользователем динамических типов не допускается;
- статическое объявление может строиться только из ограничений на типы данных; ограничения на значения данных не допускаются.

Возможности статических объявлений типов более ограничены – так, с данным синтаксисом определение квадрата как прямоугольника с двумя равными сторонами уже не представляется возможным. В некоторых случаях это ограничение можно обойти – так, квадрат можно задавать одним целым числом, описывающим сторону; в других ситуациях заменить сложные вычисления преобразованием данных будет невозможно.

3.2 Проверка типов

Для проверки типов на этапе компиляции предлагается ввести атрибуты, описывающие тип фрагментов данных для аргументов функций, параметров функций и промежуточных значений.

Формально определение метки типа описывается следующим образом:

тип = “@” пользовательский_тип | тип_атом
тип_атом = "int" | "float" | "bool" | "char" | "spec" | "delay" | "error" | "extern"

Использование составных типов в метках представляется нецелесообразным – описание сложных типов для отдельных параметров сильно усложняет прочтение кода. В тех случаях, когда составные типы использовать все же необходимо, достаточно предварительно определить их, как пользовательские статические. В приведенном ниже примере (рис. 5) функция func1 принимает один параметр типа float, возвращая Rectangle. Результат операции, возвращающей значение идентификатору data1, может быть только типа Triangle .

```
//прямоугольник задается двумя целочисленными сторонами
Rectangle << typedef @(@int,@int)
//треугольник задается тремя целочисленными сторонами
Triangle << typedef @(@int,@int,@int)
//функция func1 принимает один параметр типа float, воз-
вращая элемент типа Rectangle
func1<< funcdef@Rectangle  X@float
{
//результат операции func2, возвращающей значение иден-
тификатору data1, может быть только типа Triangle
data1@Triangle << (X,2):func2;
...
}
```

Рис. 5. Пример атрибутов статических типов

3.3 Преобразование типов

Для расширения возможностей по преобразованию типов предлагается механизм, позволяющий описывать преобразование объектов одного типа в объекты другого типа. Как и существующий, новый механизм будет использовать специальные функции, вызывающиеся при каждом явном преобразовании типов. В отличие от существующего, новый механизм не обязательно будет сохранять в преобразуемом фрагменте данных информацию о старом типе.

Заголовок преобразователя объекта типа A в объект типа B будет формально описываться следующим образом

заголовок = “transform@” имя_типа_B идентификатор@имя_типа_A

Преобразователь будет автоматически вызываться всякий раз, когда в программе будет осуществляться явное приведение объекта типа A к типу B. Так как вызов этот будет осуществляться на этапе выполнения программы, типы A и B могут быть как статическими, так и динамическими.

Использование преобразователей можно продемонстрировать на примере структур данных, описывающих полярные и декартовые координаты. Для начала работы необходимо описать типы данных для хранения координат (рис. 6).

```
//полярная координата задается двумя целыми числами –
радиусом и углом
polar << typedef @(@int,@int)
//декартовые координаты задаются двумя целыми числами –
абсциссой и ординатой
cartesian << typedef @(@int,@int)
```

Рис. 6. Описание типов координат

Несмотря на то, что оба типа данных являются списками из двух целых чисел, они различаются и не могут быть приведены друг к другу автоматически.

Первый преобразователь будет использоваться для превращения декартовых координат в полярные (рис.7).

```
transform@polar C@cartesian
{
      x << C:1;
      y << C:2;
      r << ((x,x):*,(y,y):*):+:sqrt;
      t1 << (y,x):/:abs:arctan;
      xs << (x,0):>=;
      ys << (y,0):>=;

      (t1, {(pi,t1):-}, {(pi,t1):+}, {((2,pi):*,t1):-})
>> value;
      ((xs,ys):*,(xs:-,yx):*,(xs:-,ys:-):*,(xs,ys:-
):*):? >> key;
      t << val:key:.;

      return << (r,t):polar;
}
```

Рис. 7. Преобразование декартовых координат в полярные

Второй преобразователь будет использоваться для превращения полярных координат в декартовые (рис. 8).

```
transform@cartesian C@polar
{
      r << C:1;
      t << C:2;

      x << (r,t:cos):*;
      y << (r,t:sin):*;

      return << (x,y):cartesian;
}
```

Рис. 8. Преобразование полярных координат в декартовые

В тестовой функции testCoord (рис.9) демонстрируется использование одного из преобразователей. Явное преобразование “polarCoord:cartesian” приводит к неявному вызову преобразователя (transform@cartesian C@polar). Полученный объект cartCoord имеет тип cartesian.

```
testCoord << funcdef
{
//объявление кортежа, описывающего полярную координату
      dataListPolarCoord << (10,(pi,4):/);

//преобразование объекта типа datalist в объект polar
      polarCoord << dataListPolarCoord:polar;

//явное преобразование полярных координат в декартовые
//здесь вызовется преобразователь с рис.8
      cartCoord << polarCoord:cartesian;

      return << 0;
}
```

Рис. 9. Пример использования преобразователей

Если в статических описаниях типов использование динамически описанных пользовательских типов запрещалось, в новом механизме преобразования такой запрет отсутствует. Преобразование типов осуществляется на этапе выполнения программы – и, следовательно, может работать как со статическими, так и с динамическими типами данных.

Новый формат преобразования типов обеспечивает следующие возможности:

- возможность задавать преобразователи, преобразующие объекты одних типов в другие так, чтобы они вызывались неявно, по факту обращения к обычной операции преобразования;
- возможность разделять преобразуемые объекты на элементы низкого уровня с дальнейшим формированием объектов нового типа без сохранения информации о старом типе.

4. Реализация разработанной системы типов

Представленная система типов на данный момент является теоретической разработкой; актуальной задачей является внедрение её в инструментальные средства для работы с языком “Пифагор”. Реализация предлагаемой системы типов ведет к переработке существующих инструментальных средств [12] – транслятора, генератора управляющего графа, интерпретатора и дополнительных утилит для визуализации и оптимизации.

В соответствии с новой спецификацией языка транслятор должен обрабатывать статические типы, анализируя описанные в разделах о статической типизации новые языковые конструкции. Для сохранения полученной в ходе трансляции исходной программы информации о типах оптимально будет использовать граф типов, изоморфный реверсивному информационному графу (РИГ) программы [13]. Каждая вершина графа типов будет соответствовать одной операторной вершине РИГ, храня в себе информацию о типе данных, формируемых в результате выполнения данной вершины. Хранение типов данных в отдельном графе повысит гибкость работы с инструментальной системой – как и в случае с другими слоями-графами [12], граф типов может быть необязательной опцией, дающей инструментальным средствам дополнительные возможности для анализа программы.

При формировании управляющего графа по информационному типы отдельных фрагментов данных не будут обязательной информацией, но, в определенных случаях, могут привести к объединению управляющих вершин или формированию дополнительных связей. Так, вычисления могут производиться параллельно или, напротив, строго последовательно, если исполняющей системе доступно несколько или строго одно устройство для обработки данных определенного типа. Другим примером может послужить обработка больших чисел – в одних исполняющих системах это может быть стандартной операцией, в других – автоматически преобразовываться в серию операций “длинной арифметики”.

Изменения языка также затрагивают интерпретатор, который должен автоматически отслеживать тип промежуточных значений, получающихся в ходе срабатывания отдельных операторных вершин – и, при несовпадении их с заявленными, сообщать о возникшей ошибке. Помимо этого, информация о типах может использоваться для выбора конкретного обработчика из нескольких альтернативных – и, если такая информация будет предоставлена до начала выполнения программы, к моменту вызова соответствующего обработчика он уже будет подготовлен к использованию.

Рассматривая граф типов как дополнительную надстройку над РИГ, мы предусматриваем и его отображение поверх РИГ – в существующие уже утилиты будет встроен механизм, помечающий каждую из вершин графа соответствующим ей типом. Поскольку такого рода информация может визуально перегружать изображение, её отображение представляется разумным сделать опциональным. Визуализация графа типов в виде отдельного изображения менее полезна, так как данные из него лишь дополняют данные из РИГ.

Как и в случае с генератором управляющего графа, модуль оптимизации может использовать данные о типах отдельных объектов для более грамотного упрощения информационных связей. Так, зная доподлинно о том, что в рамках функции все фрагменты данных будут исключительно скалярными значениями, можно исключить из графа все проверки на выход за границы массива.

5. Заключение

В работе рассмотрены решения, расширяющие систему типов языка “Пифагор” – статическое задание типов, статическая проверка типов, новый способ преобразования типов. Предложенные решения, выступая в качестве дополнения к динамической типизации, не только позволят упростить описание типов, но и дадут новые возможности для средств оптимизации и верификации функционально-потоковых параллельных программ.

Введение статической типизации (с сохранением динамической и сильной динамической типизаций) повысит надежность программ, позволяя решить ряд проблем однозначности до момента выполнения. Помимо этого, статическая типизация дает дополнительные возможности для оптимизации, преобразования и верификации типов.

Модификация преобразования типов, сделает работу с фрагментами данных более гибкой и, за счет упрощения структуры отдельных фрагментов данных, повысит общую эффективность работы.

Литература

1. Касьянов В. Н. Функциональный язык SISAL 3.0 / В. Н. Касьянов, Ю. В. Бирюкова, В. А. Евстигнеев // Поддержка супервычислений и Интернет-115 ориентированные технологии. – Новосибирск – 2001. – С. 54-67.
2. Odersky, M. Programming in Scala, Third Edition / Martin Odersky, Lex Spoon, Bill Venners – Artima Incorporation , USA ©2016 – ISBN 0981531687 9780981531687 – 837 p.
3. Lipovaca, M. You a Haskell for Great Good!: A Beginner's Guide / Miran Lipovaca. – No Starch Press San Francisco, CA, USA – ISBN 1593272839 9781593272838 – 2011 – 400 p.
4. Hebert F. Learn You Some Erlang for Great Good!: A Beginner's Guide / Fred Hebert. – No Starch Press San Francisco, CA, USA – ISBN 1593274351 9781593274351– 2013 – 624 p.
5. Хювёнен Э., Сеппянен И. Мир Лиспа. Т.1: Введение в язык Лисп и функциональное программирование / Хювёнен Ээро, Сеппянен Йоуко. – Пер. с финск. — М.:Мир, 1990.
6. Легалов, А. И. Функциональный язык для создания архитектурно-независимых параллельных программ / А.И. Легалов // Вычислительные технологии, № 1 (10) – Новосибирск, 2005. – с. 71-89.
7. Легалов, А. И. Использование типов в языке программирования Пифагор / А. И. Легалов, Д. В. Привалихин // В кн.: Проблемы информатизации региона. Сборник научных трудов. – Красноярск, 2002. – С. 55-61.
8. Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. / Parallel Computing Technologies, 12th International Conference PACT September-October, 2013. – St.
9. O'Sullivan B, Stewart D, Goerzen J Real World Haskell. / Bryan O'Sullivan, Don Stewart, John Goerzen – O'Reilly Media, Inc. – 2008 – Chapter 25.
10. Легалов, А. И. Эволюционное расширение пользовательских типов в языке программирования «Пифагор» / А. И. Легалов, Д. В. Привалихин // В кн.: Распределённые и кластерные вычисления. Избранные материалы третьей школы-семинара. - Красноярск, 2004. – С. 141-153.
11. Bertrand M. Eiffel: The Language / Prentice-Hall, USA – ISBN 0-13-247925-7 – 1991 – 594 p.
12. Матковский И.В Легалов А.И., Васильев В.С., Ушакова М.С Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ // Труды Института системного программирования РАН. Том 29, выпуск 5, 2017, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

13. Матковский И.В. Транслятор для функционально-потоковых параллельных программ // Я411 Языки программирования и компиляторы — 2017 : труды конференции / Южный федеральный университет ; под ред. Д. В. Дуброва. — Ростов-на-Дону : Издательство Южного федерального университета, 2017. — 282 с. ISBN 978-5-9275-2349-8