Язык программирования Smile

А. И. Легалов

14 октября 2023 г.

Оглавление

Введение

Разработка параллельных программ в настоящее время ориентирована на использование методов, учитывающих особенности архитектур целевых вычислительных систем. Это связано со стремлением повысить производительность параллельных вычислений. Исследования в области архитектурно–независимого параллельного программирования еще не сформировались до окончательных практических решений, продолжаясь по следующим направлениям:

- автоматическое или полуавтоматическое распараллеливание последовательных программ с последующей их трансформацией под целевую архитектуру [1];
- разработка программ или алгоритмов, обладающий неограниченным параллелизмом, определяемым решаемой задачей, с последующим «сжатием» этого параллелизма в соответствии с ограничениями, определяемыми целевой архитектурой [2].

При любом из этих подходов между написанной программой и реальной параллельной вычислительной системой (ПВС) существует семантический разрыв. При трансформации в машинный код происходит потеря эффективности и сбалансированности, так как характеристики «архитектурно–независимого» последовательного или параллельного алгоритма вступают в противоречие с организации вычислений в конкретных ПВС. Именно поэтому неограниченный параллелизм зачастую вручную «сжимается» при доводке программы под особенности вычислителя, определяя тем самым подход, противоположный распараллеливанию последовательных программ. Ручная трансформация ведет к потере эффективности процесса разработки параллельного программного обеспечения, не позволяя писать программу один раз и для различных архитектур. В связи с этим актуальной является задача поиска моделей параллельных вычислений и построение на их основе языковых и инструментальных средств, обеспечивающих эффективную подстройку параллелизма однажды написанной программы под различные вычислительные ресурсы.

Одним из путей, определяющим возможность более эффективной трансформации к реальным архитектурам, является использования в языках программирования статической типизации данных, которая обеспечивает эффективную компиляцию в бестиповое представление на уровне системы команд [3]. Использование этого подхода широко распространено как в последовательном, так и в параллельном программировании. Однако, для написания архитектурно-независимых параллельных программ только статической типизации недостаточно. Необходимо также учитывать особенности конструкций, описывающих параллелизм, динамические характе-

ристики которых могут затруднять трансформацию в машинный код существующих ПВС.

От автора

Концепция архитектурно-независимого параллельного программирования (АНПП) на текущий момент не находит приверженцев. Мои прогнозы относительно путей развития методов параллельного программирования, сформулированные в конце девяностых, не оправдались. По прошествии практически уже почти тридцати лет разработка параллельных программ остается привязанной к особенностям конкретных параллельных вычислительных систем (ПВС). Меняются архитектуры как последовательных, так и параллельных компьютеров, но все также неизменным остается подход, направленный на написание программ, ориентированных для каждой из таких систем и не рассматривающий общую теорию параллельных программ в качестве первоосновы.

В настоящее время очень мало работ в области анализа особенностей построения параллельных программ как некоторых универсальных конструкций, напрямую не связанных с реальными вычислителями. Среди них следует отметить работы Алеевой В.Н. [4], представляющих параллельные вычисления на основе Q-детерминанта. Также близкую тему затрагивают работы по ресурсно-независимому параллельному программированию, ориентированному в то же время на спецвычислители, разработанные с применением ПЛИС [5]. Но больше ничего конкретного назвать не могу. Поэтому остается верить, что то, чем в рамках научных исследований я занимаюсь, не является тупиковой ветвью и когда-нибудь (желательно как можно быстрее, пока я нахожусь в ясном уме и трезвой памяти) выстрелит.

AL: Скорее всего в дальнейшем этот обзор расширится но не сильно

Опираясь на эту веру, я продолжаю развивать концепцию архитектурно-независимого параллельного программирования на основе функционально-потоковой парадигмы, основной идеей которой является написание программ совершенно не связанных с ресурсными ограничениями реальных ПВС. В рамках этой концепции ключевым аспектом является эффективная трансформация архитектурно-независимых параллельных программ в уже существующие архитектуры, а не их непосредственное выполнение на специально создаваемых системах. На текущий момент разработан язык функционально-потокового параллельного программирования (ЯФППП) Пифагор [6]. Однако в его основе лежит динамическая система типов, что не обеспечивает гибкой трансформации в реальные параллельные архитектуры. Как и для любого динамически типизированного языка в данном случае имеются много накладных расходов, ведущий к резкому уменьшению производительности вычислений. Помимо этого операторы функционально-потоковой модели параллельных вычислений (ФПМПВ) в большинстве своем ориентированы на динамическую организацию данных и хранение элементов произвольного типа, что также является ограничивающим фактором для преобразований архитектурно-независимых параллельных программ в ресурсно-зависимую форму, характерную для современ-

ных вычилительных систем (ВС). Поэтому для дальнейшего продолжения работ необходимо сформировать концепцию модели вычислений, которая бы обеспечила поддержку более эффективных трансформаций в архитектуры реальных ПВС для создаваемого на основе этой модели языка программирования.

Подобные концепции в настоящее время обычно опираются на статическую типизацию и организацию данных, размерность которых и внутренняя структура в основном определяются на этапе компиляции. Внедрение этих идей требует пересмотра ряда понятий лежащих в основе уже разработанных ФПМПВ и ЯФППП. Поэтому необходима разработка другой модели, ориентированной на поддержку новых идей которая бы обеспечила основу для создания соответствующего языка. Ниже рассматриваются концепции статически типизированной модели функционально-потоковых параллельных вычислений (СТМФППВ), которую предполагается положить в основу соответствующего статически типизированного ЯФППП (СТЯФППП) Smile. Название языка выбрано исходя из давно возникших ассоциаций, что многие из формируемых в процесс написания программ сочетаний символов образуют комбинации, напоминающие смайлики. Это определило выбор смайлика «:-)» в качестве символа языка.

О чем эта книга

Данный план описывает приблизительное содержание работы. В ходе написания он может изменяться. Пока же он представляет некоторый аналог персонального мозгового штурма.

1. Статически типизированная модель модель функционально-потоковых параллельных вычислений.
   a) Базовые положения.
   b) Общие принципы организации модели.
2. Язык программирования Smile. Предварительное описание.
3. Семантика оператора интерпретации.
4. Синтаксис языка программирования Smile.

1 Особенности архитектурно-независимой парадигмы параллельного программирования

Поддержка архитектурной независимости на уровне моделей вычислений и языков программирования, связана с подходами, обеспечивающими специфические методы хранения данных, формируемых в ходе выполнения программы, а также независимостью стратегии управления вычислениями [7] от реальных вычислительных ресурсов.

Независимость хранения данных от памяти поддерживается функциональной парадигмой программирования, ориентированной на представление программ в виде взаимодействующих функций. В отличие от императивного подхода память данных представлена в неявном виде. Использование рекурсий вместо итераций позволяет избавиться на уровне описания алгоритмов от повторного использования переменных. Эти решения во многом обеспечивает архитектурную независимость программ и реализованы в разнообразных языках функционального программирования (ЯФП). Однако большинство таких языков имеют ограничения по неявному представлению параллелизма задачи, что обуславливается особенностями представления структур данных в виде списков с последовательным доступом к их элементам. Наличие в списке только варианта доступа к голове и хвосту не позволяет организовать параллельные вычисления непосредственно на основе текущих реализаций. Поэтому для поддержки параллелизма в ЯФП обычно используется явное управления, на основе которого создаются потоки или процессы, что ведет к произвольному воздействию на параллелизм со стороны разработчика и является фактором, определяющим наличие архитектурной зависимости.

Стратегии управления вычислениями по готовности данных (dataflow control) позволяют неявно описывать параллелизм. Одной из первых таких моделей вычислений (МВ) является модель Денниса [8]. Она легла в основу ряда специализированных процессоров с различной архитектурой. Можно отметить различные языковые средства, использующие управление по готовности данных, и применяемых для программирования различных архитектур. Например: Sisal [9], Colamo [10], LuNA [11]. В ряде систем программирования управление потоками данных сочетается с функциональным стилем. Однако для многих МВ проблематично говорить об архитектурной независимости, что зачастую связано с ориентацией языков

программирования и методов их трансформации на определенные архитектурные решения. В большинстве из них до конца не проработана концепция неограниченного параллелизма. Также часто управление по готовности данных сочетается с использованием явного управления или с необходимостью управления ограниченными ресурсами.

Наряду с функциональным подходом и управлением по готовности данных ряд задач, связанных с архитектурно-независимым представлением параллелизма, можно решить, используя специальные структуры, которые не только содержат данные, но и обеспечивают поддержку их разнообразного параллельного поведения. При этом отличия в поведении вводимых конструкций определяет подходы к различной организации параллелизма. Инкапсуляция динамического поведения данных внутри специальных структур (динамически формируемых данных) позволяет убрать из программ явное управление вычислениями, обычно применяемое в императивных языках программирования, заменяя его на взаимодействие с функциями и другими структурами неявным управлением по готовности данных. Применение параллельной рекурсии позволяет рассматривать программу как описание активностей, выполняемых в неограниченных вычислительных ресурсах, формируемых неявно по мере надобности. Подобный подход на уровне языка программирования позволяет использовать его в качестве архитектурно–независимого. Вместе с тем это предъявляет особые требования к трансформации программ с таких языков в архитектурно–зависимые программы.

Описанный подход был предложен при разработке языка функционально–потокового параллельного программирования Пифагор, разработанного на основе функционально–потоковой модели параллельных вычислений (ФПМПВ) [12, 13]. Соответствующие структуры представлены в нем как списки данных, параллельные списки, задержанные списки, асинхронные списки. Каждый вид списков задает свои методы группировки данных и способы управления по готовности этих данных. К недостаткам предложенных конструкций, как и языка в целом можно отнести динамическую типизацию атомарных типов, а также динамическое формирование списков в ходе вычислений, что не позволяет во время компиляции программы сформировать эффективное выходное представление.

Необходимость использования статической типизации для повышения эффективности трансформации в реальные архитектуры была подтверждена в ходе реализации ряда проектов по трансформации функционально-потоковых параллельных программ:

- при преобразовании в топологию ПЛИС [14];
- при трансформации в статически типизированный императивный язык программирования [15].

Для получения требуемых решений в промежуточное представление, порождаемое компилятором языка Пифагор, пришлось вводить дополнительные описания, определяющие типы обрабатываемых данных.

Таким образом, для решения задачи эффективной трансформации архитектурно–независимой параллельной программы в программу для реальной целевой архитектуры необходимо совместно использовать подходы, которые по отдельности не решают целевую задачу:

1. неявное управление вычислениями по готовности данных (dataflow control);
2. функциональную парадигму программирования;
3. специальные структуры данных, ориентированные на представление различных видов параллелизма;
4. статическую типизацию данных.

Их совместное использование отражается как на модели параллельных вычислений, так и инструментальных средствах, создаваемых на ее основе. Ориентация на архитектурную независимость по сути ведет к формированию предметно-ориентированной модели параллельных вычислений и создаваемому на ее основе предметно ориентированному языку архитектурно-независимого параллельного программирования со специфическим для него набором артефактов и их семантикой.

2 Статически типизированная модель модель функционально-потоковых параллельных вычислений

> **Примечание**: Описывается модель параллельных вычислений, ориентированная на представление динамики поведения при статической типизации. То есть речь идет о статически типизированной модели функционально-потоковых параллельных вычислений (СТМФППВ). Ее отличительной чертой является более эффективная поддержка процесса трансформации программ во время компиляции.
> *Предполагается, что в данном тексте замечания могут касаться и изменений, вносимых в модель и язык. В ходе дальнейших версий эти замечания будут удаляться.*

2.1 Базовые принципы

Статически типизированная модель модель функционально-потоковых параллельных вычислений (СТМФППВ) определяет базовую семантику и динамику поведения статически типизированного языка функционально-потокового параллельного программирования (СТЯФППП) Smile. В отличие от ранее предложенной функционально-потоковой модели параллельных вычислений (ФПМПВ) данная модель ориентирована на применение статической системы типов и фиксированные размерности контейнерных данных, используемых в массовых операциях. Это, в свою очередь, ведет к изменению семантики программо-формирующих операторов. Изменяются также аксиомы модели и ее алгебра преобразований за счет ориентации на период компиляции. Вместе с тем основные характеристики модели, определяющие концепцию архитектурно-независимого параллельного программирования остаются практически неизменными:

- вычисления протекают внутри неограниченных ресурсов, что позволяет неявно описывать параллелизм без возникновения ресурсных конфликтов;
- управление вычислениями осуществляется по готовности данных;

- выбор операций и аксиом, определяющих базовый набор функций, ориентирован на наглядное текстовое представление информационного графа программы при его описании на языке программирования;

- модель вычислений определяет общую структуру функционально-потоковой параллельной программы без привязки к операционной семантике, которая может определяться дополнительно, определяя тем самым специфику конкретного языка (подъязыка) функционально-потокового параллельного программирования.

> **Примечание**: *Начинает создаваться впечатление, что на уровне модели вычислений использование статической системы типов для описания базовых типов данных является избыточным. Это связано с тем, что конкретика в данных не особо влияет на семантику, связанную с управлением вычислением. В большей степени это проявляется на уровне языка программирования. Поэтму имеет смысл попытаться пересмотреть описание модели и языка именнос этих позиций.*

Первое требование обеспечивает ресурсную независимость предлагаемой модели, что позволяет описать параллелизм, ограниченный только информационными отношениями между функциями и данными, присущими решаемой задаче. Это сводит перенос разработанной и отлаженной функционально-потоковой параллельной программы на любую вычислительную систему к распределению ресурсов в соответствии с целевой архитектурой. Подобный подход используется также в ряде известных схем потока данных (СПД) [Алгоритмы1982], ориентированных на рекурсивное описание программ, обрабатывающих только один входной поток данных и не поддерживающих их конвейерное продвижение. В связи с отсутствием циклических конструкций граф данной модели является ациклическим.

Использование текстового представления параллельных программ связано с трудностями непосредственного описания информационного графа, что привело к синтаксису языка, несколько отличающемуся от общепринятого. Кроме того, в языке отсутствуют вентили, обеспечивающие условную передачу данных в традиционных СПД [Деннис1972, Arvind1975]. Эти вентили трудно структурировать при текстовом описании программ без использования дополнительной синхронизации информационных потоков.

Отличительной особенностью СТМФППВ является ориентация программо-формирующих операторов на использование статически типизированных вычислений. Это ведет к уменьшению динамических свойств операторов, что облегчает трансформацию в структуры данных, типичные практически для всех современных статически типизированных языков программирования (как императивных, так и функциональных). Подобные модификации, в свою очередь, достаточно сильно изменяет семантику как модели вычислений, так и формируемого на ее основе СТЯФППП Smile.

> **Примечание**: Следует отметить, что понятие СТМФППВ не связано только со статической типизацией данных. Как и основная ФПМПВ новая модель предназначена для описания семантики вычислений и организации параллельных процессов. Вместе с тем следует отметить, что использование статической типизации накладывает определенную специфику на операторы модели, что, в свою очередь, ведет к изменению их семантики функционирования. В связи с этим поведение операторов модели и, как следствие, программы, написанной на соответствующем языке, тоже будет иным. Изменяются аксиомы модели и алгебра эквивалентных преобразований.

2.2 Общие принципы организации модели

Модель задается тройкой:

$$\mathbf{M} = (\mathbf{G}, \mathbf{P}, \mathbf{S_0}),$$

где $\mathbf{G}$ — ациклический ориентированный граф, определяющий информационную структуру программы (ее информационный граф), $\mathbf{P}$ — набор правил, определяющих динамику функционирования модели (механизм формирования разметки), $\mathbf{S_0}$ — начальная разметка дуг графа, на которых уже сформированы данные, определяющие динамику выполнения.

Информационный граф:

$$\mathbf{G} = (\mathbf{V}, \mathbf{A}),$$

где $\mathbf{V}$ — множество вершин определяющих программоформирующие операторы, а $\mathbf{A}$ — множество дуг, задающих пути передачи информации между ними.

Вершины графа, соответствующие программоформирующим операторам и хранилищам, обеспечивают информационные преобразования данных, их структуризацию и размножение. Существуют следующие типы вершин:

- операторы интерпретации;
- константные операторы (константы);
- оператор копирования (обозначение);
- операторы группировки в списки (составные структуры);
- оператор задержки (задержка).

Использование в СТМФППВ статической типизации вместо динамической накладывает, с одной стороны, свои ограничения, но, с другой стороны, предоставляет дополнительные возможности по трансформации функционально–потоковых параллельных программ в программы для реальных архитектур.

2.2.1 Операторы интерпретации

Операторы интерпретации предназначены для описания функциональных преобразования аргументов. Каждый такой оператор имеет два входа, на один из которых поступает значение, интерпретируемое как функция **F** (функциональный вход), а на другой величина, являющаяся аргументом (вход данных) **X**, обрабатываемым данной функцией. Использование статической типизации, а также изменение семантики списков в новой модели привели к разделению оператора интерпретации на два разных вида: одиночный (одноаргументный) и групповой (массовый, поэлементный).

Одиночный оператор интерпретации, обозначаемый в текстовом представлении, как и в ФПМПВ [12], через «:» (постфиксная форма) или «^» (префиксная форма), предназначен для задания обычных функций, воспринимающих аргумент в качестве единого целого.

Массовый оператор интерпретации используется для задания вычислений над каждым элементом списка, порождая на выходе контейнер с элементами тип которых соответствует типу результата выполняемой функции. Обозначается двойным значком «::» для постфиксной или «^^» для префиксной форм соответственно. В дальнейшем для иллюстрации в примерах используются только постфиксные формы операторов интерпретации.

Использование разных обозначений позволяет однозначно применять функцию с одним и тем же именем в разных контекстах. Например функция вычитания «-» над аргументом `(10, -3)`, воспринимаемом как вектор, состоящий из двух целых чисел, порождает следующие значения:

- $(\mathbf{10}, -\mathbf{3}) : - \Rightarrow \mathbf{13}$ — двухместная функция вычитания над одним аргументом;
- $(\mathbf{10}, -\mathbf{3}) :: - \Rightarrow (-\mathbf{10}, \mathbf{3})$ — групповая функция смены знака.

Разделение оператора интерпретации на одиночный и групповой позволяет ввести гибкий набор дополнительных функций для списков различной структуры, обеспечивая при этом более разнообразную обработку асинхронно поступающих данных.

2.2.2 Константный оператор

Константный оператор или **константа** определяет вершину информационного графа, хранящую постоянную величину и всегда готовую к выполнению. Данный оператор не имеет входа. На выходе изначально устанавливается разметка, определяющая предписанное значение. Множество константных операторов информационного графа формируют внутреннюю начальную разметку модели вычислений. В языковом представлении константный оператор задается значением соответствующего типа. Тип константы должен быть известен во время компиляции функции. К константам относятся данные различных типов, например: целые, булевские, сигналы, атомарные функции и функции, определенные программистом. Отнесение к

константам функций обуславливается тем, что после их описания они зафиксированы и предоставляют данные непосредственно готовые к выполнению. Примеры констант:

- **10** — целочисленная константа;
- **true** — булевская константа;
- **!** — сигнальная константа;
- **+** — атомарная функция плюс;
- **min** — имя разработанной функции нахождения минимума двух элементов.

2.2.3 Оператор копирования

Оператор копирования представляет собой узел, осуществляющий передачу данных, поступающих на его единственный вход, на множество выходов. По сути это источник данных к которому осуществляется доступ из связанных с ним узлов. Поэтому представленное отдельное графическое обозначение не означает реально выделенного отдельного оператора. В общем случае оператор копирования может объединяться с предшествующим оператором, из которого выходит его выходная дуга. Также возможна цепочка операторов копирования, которая может восприниматься как один оператор. В текстовой форме он определяется через именование передаваемой величины и дальнейшее использование введенного обозначения в требуемых точках функции. Используются как постфиксное именование размножаемого объекта в форме: `величина >> имя`, так и его префиксный эквивалент, имеющий вид: `имя << величина`. Например:

```
y << F^x;
(x,y):+ >> c;
```

Тип обозначения совпадает с типом результата предшествующих вычислений и определяется во время компиляции.

2.2.4 Операторы группировки в списки

Операторы группировки в списки (или списки) предназначены для структурирования и формирования поведения поступающих в них данных различными способами, что и определяет разнообразие описываемых вариантов параллелизма, задаваемого моделью вычислений. По формируемому поведению можно выделить следующие группы списков:

- синхронный список или соединитель (join), обеспечивающий сбор всех поступающих данных воедино до последующей их выдачи на выход;

- параллельный список или рой (swarm), осуществляющий упорядочивание по номеру входа, но не синхронизирующий данные поступающие на различные входы;
- асинхронный список или поток (stream), осуществляющий упорядочение данных в соответствии с временем их поступления и выдающий эти данные последовательно в порядке поступления.

Каждый вид списка характеризуется как своим типом, так и типом или типами его элементов, а также числом элементов размещаемых в списке. Следует при этом отметить, что эти параметры для каждого вида списков могут отличаться по допустимым значениям.

Соединитель

Соединитель (join) имеет несколько входов и один выход. Он обеспечивает структуризацию, упорядочение и синхронизацию данных, поступающих по входным дугам из различных источников. Типы поступающих элементов должны быть известны на этапе компиляции. Порядок элементов определяется номерами входов, каждому из которых соответствует натуральное число в диапазоне от $\mathbf{0}$ до $\mathbf{N-1}$, где $\mathbf{N}$ — длина формируемого списка. Соединитель готов к последующей обработке, когда в него поступят все входные данные. В текстовом виде оператор задается ограничением элементов списка круглыми скобками «(» и «)». Например:

$$(\mathbf{x1}, \mathbf{x2}, \mathbf{x3}, \mathbf{x4})$$

Нумерация элементов списка начинается с нуля и задается неявно в соответствии с порядком их следования слева направо. Тип элементов списка должен быть известен во время компиляции и определяет одну из возможных вариантов его интерпретации: вектор или кортеж.

Соединители, имеющие тип вектор (**vector**) предназначены для группировки элементов одного типа. Это позволяет обращаться к ним по имени, а также задавать над ними массовые операции, обеспечивающие параллелизм.

Списки, образующие кортеж (**tuple**), также имеет несколько входов и один выход. Он обеспечивает структуризацию, упорядочение и синхронизацию разнотипных данных, поступающих по дугам из различных источников. Типы поступающих данных должны быть известны во время компиляции. Доступ к элементам осуществляется по порядковому номеру (индексу). В текстовом виде задается ограничением элементов списка круглыми скобками «(» и «)» аналогично вектору.

Рой

Рой (swarm), в отличие от соединителя, группирует независимые друг от друга данные. Поступление в рой каждого элемента сопровождается выдачей сигнала готовности, информирующих об этом событии, узлы информационного графа, принимающие от него информацию. Это позволяет оперативно и асинхронно реагировать на изменение состояния роя.

В текстовом виде группировка в рой задается ограничением его элементов квадратными скобками «[» и «]». Например:

$$[\mathbf{x1}, \mathbf{x2}, \mathbf{x3}, \mathbf{x4}]$$

Каждый элемент роя формируется независимо и по его появлению готов к выполнению. Как и у соединителя элементы роя могут быть однотипными, образуя вектор, или разнотипными, образуя кортеж.

Поток

Понятие потока (**stream**) расширяет концепцию ранее предложенного асинхронного списка [13]. Основная идея, связанная с асинхронным поступлением данных, сохраняется. Однако предполагается, что все элементы имеют один и тот же тип, который, в свою очередь, не может являться потоком или роем. Это вполне соответствует концепциям универсальных статически типизированных языков. Поток можно рассматривать как сущность к основным характеристикам которой относятся:

- при появлении в потоке хотя бы одного готового элемента данных, он порождает сигнал, информирующий об его готовности;
- готовый элемент может быть прочитан из потока для обработки;
- если во время обработки элемента, выбранного из потока в него поступают новые элементы данных, они также могут асинхронно выбираться из потока в порядке поступления и обрабатываться параллельно;
- параллельно обрабатываемые элементы потока могут поступать после обработки в другой поток, тип которого определяется типом результата функции, при этом порядок их поступления может отличаться от первоначального в зависимости от времени обработки;
- поток можно проверить на отсутствие дальнейшего поступления данных, что позволяет завершить работу с ним.

2.2.5 Задержка

Оператор задержки или задержка задается вершиной, содержащей допустимый информационный подграф, в который входят несколько входных дуг и выходит одна выходная дуга. Входные дуги определяют поступление аргументов, а выход задает выдаваемый из подграфа результат. Специфической особенностью такой группировки является то, что ограниченные оператором задержки вершины, представляющие другие программоформирующие операторы, не могут выполняться, даже при наличии на входах всех аргументов. Их активизация возможна только при снятии задержки (раскрытии контура), когда ограниченный подграф становится частью всего вычисляемого графа.

Первоначально задержанный подграф создает на своем единственном выходе константную разметку, которая является образом (иконкой) данного подграфа. Эта разметка распространяется по дугам графа от одного программоформирующего оператора к другому, размножаясь, входя в различные списки и выделяясь из них до тех пор, пока не поступит на один из входов оператора интерпретации. Как только оператор задержки становится одним из аргументов оператора интерпретации, вместо иконки происходит подстановка ранее определенного задержанного подграфа с сохранением входных связей. Опоясывающий подграф контур оператора задержки при этом «убирается», и происходит выполнение активированных операторов. В результате на выходной дуге раскрытого подграфа вновь формируется результирующая разметка, которая и является одним из аргументов оператора интерпретации, раскрывшего задержанный подграф. Данная процедура называется раскрытием задержанного подграфа.

В текстовом виде оператор задержки задается охватом других операторов фигурными скобками «{» и «}». Например:

$$\{(\mathbf{a}, \mathbf{b}) : +\}$$

Если внутри задержки необходимо сформировать несколько независимых аргументов, то они группируются в рой, который инициируется при раскрытии:

$$\{[\mathbf{x1}, \mathbf{x2}, \mathbf{x3}, \mathbf{x4}]\}$$

Наличие этой конструкции позволяет откладывать момент начала некоторых вычислений или вообще не начинать их, что необходимо при организации выборочной обработки данных. Помимо этого данный оператор, при необходимости, может использоваться в качестве скобок, меняющих приоритет выполнения операторов. Для этого он может быть непосредственно представлен как один из аргументов оператора интерпретации.

2.3 Отношения между операторами группировки и интерпретации

Вычисления формируются при поступлении данных на операторы интерпретации, каждый из которых принимает два операнда, один из которых является функцией, а другой аргументом. В ходе интерпретации формируется результат вычислений тип которого зависит как от типов аргумента и функции, так и от типа операции интерпретации. Разнообразные комбинации этих трех составляющих образуют операционную семантику модели вычислений, а также определяют варианты эквивалентных трансформаций в соответствии с алгеброй модели.

2.3.1 Использование одиночного оператора интерпретации

Одиночные оператор интерпретации в большинстве случаев определяет традиционные функциональные преобразования своих аргументов в результат. При этом

допустимы следующие комбинации отношений между аргументами одиночного оператора интерпретации:

- скаляр — скаляр;
- соединитель — скаляр;
- рой — скаляр;
- поток — скаляр;
- скаляр — соединитель;
- соединитель — соединитель;
- рой — соединитель;
- поток — соединитель;
- скаляр — рой;
- соединитель — рой;
- рой — рой;
- поток — рой;

В представленных отношениях первый аргумент выступает в роли обрабатываемых данных, а второй определяет функцию, осуществляющую обработку этих данных. Следует отметить, что поток не может непосредственно выступать в роли функции. Аргумент является скаляром, если он имеет предопределенный базовый (атомарный) тип или является константой одного из базовых типов.

Отношение «скаляр — скаляр»

Данное отношение воспринимается одиночным оператором интерпретации как традиционная функция от одного аргумента. То есть, обрабатываемые данные с константам или вычисляемым значениям базового типа. Формируемый результат определяется семантикой данных, интерпретируемых как функция. Функция может быт как предопределенной, так и разработанной программистом. Например:

$$\mathbf{x} : - \equiv -\mathbf{x}$$
$$\mathbf{x} : \mathbf{sin} \equiv \mathbf{sin}(\mathbf{x})$$

Отношение «соединитель — скаляр»

Отношение практически аналогично выполнению функции от нескольких аргументов, значения который определяются как элементы соединителя:

$$(\mathbf{x}, \mathbf{y}) : + \equiv +(\mathbf{x}, \mathbf{y}) \equiv \mathbf{x} + \mathbf{y}$$
$$(\mathbf{x}, \mathbf{y}) : \mathbf{min} \equiv \mathbf{min}(\mathbf{x}, \mathbf{y})$$

Отношение «рой — скаляр»

Данное отношение позволяет формировать асинхронное поступление аргументов в функцию, осуществляющую их отображение. Предполагается, что при появлении любого элемента в рое оператор интерпретации запустит функцию, осуществляющую обработку роя, которая осуществит частичное вычисление. То есть в данном случае отсутствует предварительная синхронизация аргументов перед вызовом функции. Ситуация во многом аналогична использованию вместо функции **inline**–подстановки. Предполагается что такие вычисления возможны, если при описании разрабатываемой функции ее входной параметр описан как рой. Пример:

$$[\mathbf{x}, \mathbf{y}] : \mathbf{min}$$

Отношение «поток — скаляр»

Данное отношение передает на вход функции, запускаемой оператором интерпретации поток, готовность которого определяется появлением первого элемента. Определение готовности данных для последующих элементов поступившего потока формируется уже внутри функции осуществляющей его обработку.

Отношения «данные — соединитель>

Соединитель в качестве аргумента–функции оператора интерпретации задает одновременное выполнение всех указанных в нем функций над обрабатываемым аргументом. То есть, в данном случае реализуется параллелизм с множеством независимых потоков команд над одним набором данных. Для любых входных данных $\mathbf{X}$ и списка функций $\mathbf{F} = (\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}})$ в ходе интерпретации осуществляется эквивалентное преобразование в набор параллельно исполняемых операторов:

$$\mathbf{X} : (\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}) \equiv (\mathbf{X} : \mathbf{F_0}, \mathbf{X} : \mathbf{F_1}, ..., \mathbf{X} : \mathbf{F_{n-1}})$$

На выходе формируется результат–соединитель.

Отношения «данные — рой>

Варианты с использованием в качестве набора функций роя во многом схожи с применением в этой роли соединителя. Отличие проявляется в том, что его аргументы могут вычисляться и в последующем обрабатывать входной аргумент без общей синхронизации. Поэтому появление результатов на отдельных выходах оператора интерпретации, реализующего данное отношение, может быть произвольным с формированием нового роя. Для входных данных $\mathbf{X}$ и списка функций $\mathbf{F} = [\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}]$ в ходе интерпретации осуществляется эквивалентное преобразование в набор параллельно исполняемых операторов:

$$\mathbf{X} : [\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}] \equiv [\mathbf{X} : \mathbf{F_0}, \mathbf{X} : \mathbf{F_1}, ..., \mathbf{X} : \mathbf{F_{n-1}}]$$

На выходе формируется результат–рой.

2.3.2 Использование группового оператора интерпретации

Групповой оператор интерпретации ориентирован на массовую обработку множества структурированных наборов данных одной командой. Основной его задачей является задание поэлементной обработки различных списков. При этом окончательные вычисления формируются через одиночные операторы интерпретации, сведение к которым осуществляется за счет алгебры эквивалентных преобразований. Для группового оператора интерпретации допустимы следующие комбинации отношений:

- соединитель — скаляр;
- рой — скаляр;
- поток — скаляр;

В отличие от одиночного оператора интерпретации в групповом операторе ведется выполнение функции над элементами группы. Более сложные варианты функций, образующие различные списки, ведут к формированию многомерных конструкций и на уровне модели вычислений пока не рассматриваются.

Отношение «соединитель — скаляр»

В рамках этого отношения соединитель рассматривается как вектор, состоящий из однотипных элементов, к каждому из которых применяется одна и та же функция, выступающая в роли скаляра.

Пусть $\mathbf{X} \equiv (\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}})$ — соединитель, задающий вектор обрабатываемых данных, $\mathbf{f}$ — функция. Тогда, с учетом последующих эквивалентных преобразований, групповой оператор интерпретации для данного отношения можно представить следующим образом:

$$\mathbf{X} :: \mathbf{f} \equiv (\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}) :: \mathbf{f} \equiv$$
$$\equiv (\mathbf{x_0} : \mathbf{f}, \mathbf{x_1} : \mathbf{f}, ..., \mathbf{x_{n-1}} : \mathbf{f})$$

Отношение «рой — скаляр»

Это отношение во многом аналогично предыдущему. Отличие заключается в том, что выполнение функций над отдельными элементами начинается сразу же по поступлении этих элементов. Также асинхронно формируется рой в результате выполнения функций.

Рой задается следующим образом: $\mathbf{X} \equiv [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}]$. Групповая операция над роем с использованием функции $\mathbf{f}$ определяется следующим эквивалентным преобразованием:

$$\mathbf{X} :: \mathbf{f} \equiv [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}] :: \mathbf{f} \equiv$$
$$\equiv [\mathbf{x_0} : \mathbf{f}, \mathbf{x_1} : \mathbf{f}, ..., \mathbf{x_{n-1}} : \mathbf{f}]$$

Отношение «поток — скаляр»

Групповая интерпретация отношения осуществляется по аналогии с предшествующими. То есть, функция выполняется над каждым элементом потока до тех пор, пока в поток из различных источников поступают элементы. В результате на выходе формируется новый поток. Следует отметить, что порядок следования результатов в новом потоке может не совпадать с порядком следования исходных данных. Это объясняется тем, что даже при выполнении одной и той же функции время вычислений может отличаться по разным причинам. Спецификой потока является также то, что количество обрабатываемых элементов может быть неизвестным заранее, а завершение поступления элементов определяется по считыванию маркера конца потока.

3 Статически типизированный язык функционально–потокового параллельного программирования Smile

Статически типизированный язык функционально-потокового программирования Smile разрабатывается с учетом опыта, полученного при создания языка программирования Пифагор [6]. Его основным отличием является использование статической системы типов, обеспечивающей эффективный анализ и трансформацию на этапе компиляции. Предполагается, что должно улучшиться распределение памяти под хранимые данные, а также генерация кода в другие языки программирования со статической типизацией, ориентированные на написание как параллельных, так и последовательных программ с использованием императивного или функционального программирования.

Опираясь на СТМФППВ [20], разрабатываемый язык программирования включает дополнительные конструкции, расширяющие его функциональные возможности по сравнению с описываемыми моделью вычислений. Это обуславливается тем, что, в отличие от модели вычислений, язык программирования должен обеспечивать поддержку удобного и эффективного написания программ.

3.1 Используемый метаязык

При описании синтаксиса языка используются расширенные формы Бэкуса–Наура (РБНФ). Квадратные скобки «[» и «]» означают, что заключенная в них сентенциальная форма может отсутствовать, фигурные скобки «{» и «}» означают ее повторение (возможно, 0 раз), а круглые скобки «(» и «)» используются для ограничения альтернативных конструкций. Сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один или более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл, написанными на русском языке и разделенными, при необходимости, знаком подчеркивания «_» Каждое правило оканчивается точкой «.». Терминальные символы изображаются словами, написанными строчными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки.

Левая часть правила отделяется от правой знаком «=».

3.2 Элементарные конструкции

Элементарные конструкции являются составными частями различных других конструкций, образующих на верхнем уровне артефакты. К артефактам относятся сформированные программные объекты, обладающие определенной семантикой и используемые для описания как различных данных, так и функций.

3.2.1 Разделители

Пробелы, символы табуляции, перевода на новую строку и перевода страницы используются как разделители. Вместо одного из таких символов может использоваться любое их количество. Все другие управляющие символы употреблять в тексте программы запрещено.

Комментарии

Многострочные комментарии начинаются парой символов «/*» и заканчиваются парой символов */. Разрешены везде, где возможны разделители. Вложенность многострочных комментариев не допускается.

В языке также допускаются однострочные комментарии. Они начинаются парой символов "//"и заканчиваются признаком конца строки. Однострочные комментарии могут начинаться с самого начала строки или стоять после операторов, написанных в этой строке.

```
комментарий = "/*" {знак} "*/" | "//" {знак}.
```

Примеры:

```
/* Многострочный комментарий в одной строке */

/*
 * Многострочный комментарий,
 * размещенный в нескольких строках
 */

// Однострочный комментарий
```

3.2.2 Идентификаторы

Идентификаторы используются для обозначения имен констант, переменных, функций и типов данных. Допустимые символы: цифры 0-9, прописные и строчные буквы латинского алфавита A-Z, a-z, символ подчеркивания «_». Первый символ не

является цифрой. Идентификатор может быть произвольной длины. Прописные и строчные буквы различаются.

```
ид := (буква | "_"){буква|цифра|"_"}.
```

Примеры:

```
NAME1
name1
it_is_ID
```

3.2.3 Зарезервированные слова

Зарезервированные слова используются для ключевых слов встроенных типов данных, предопределенных обозначений и функций. Ниже приведен общий их список:

```
block       break       bool        char        const
group       dup         double      delay       else
error       false       func        funcdef     int
nil         queue       return      signal      swarm
transform   true        type        vector
```

Зарезервированные слова записываются строчными буквами. Использовать их в качестве идентификаторов запрещено.

Примечание. *Следует отметить, что типы данных, предопределенные функции будут вынесены в отдельную категорию предопределенных артефактов. То есть предполагается формирование данных артефактов в соответствующем пространстве имен, запрещающем их повторное использование в другом контексте. Но это будет описано позднее, когда устаканится их резервирование в данной роли. Также возможно дальнейшее расширение набора зарезервированных слов.*

3.2.4 Специальные знаки

Специальные знаки используются для обозначения различных понятий. Они могут являться разделителями, элементами программоформирующих операторов, обозначениями констант данных и предопределенных функций. Их смысл будет раскрыт при описании семантики языка.

```
спецзнак =
    ":"  | "::" | "^"  | "^^" | "+"  | "-"  | "/"  | "*"  |
    "%"  | "/%" | "<"  | ">"  | "="  | ">=" | "<=" | "!=" |
    "=>" | "->" | "<-" | "()" | "{}" | "[]" | "|"  | "#"  |
    ".." | "?"  | "??" | "!".
```

Примечание. *Следует отметить, что ряд специальных знаков в настоящее время не используется и зарезервирован для дальнейшего расширения языка. Перечисленный набор спецзнаков в дальнейшем может корректироваться. Также в ходе уточнения описания будет определяться и их принадлежность к конкретному типу. В целом предполагается, что большинство спецзнаков в контексте вычисления будут отображаться в функции. Вместе с тем, дуализм аргументов операции интерпретации может привести к тому, что ряд знаков будет интерпретироваться различным способом в зависимости от того, являются они данными или функциями в операторе интерпретации.*

3.3 Обозначения

В языке, основанном на принципе на сочетании принципов единственного использования вычислительных ресурсов и единственного присваивания, отсутствует понятие переменной. Вместо этого вводится понятие обозначения как идентификатора, поставленного в соответствие с каким-либо программным фрагментом. В пределах некоторой области видимости использование идентификатора в качестве обозначения должно быть уникальным. В ходе описания обозначение получает тип. Помимо этого оно связывается с величиной (значение), задаваемым либо константой, либо полученным в ходе вычислений. Эти параметры обозначения могут использоваться для дальнейшей передачи в любую точку программы, обеспечивая тем самым копирование артефакта, полученного в ходе вычислений. В языке определены следующие способы задания обозначений:

- префиксное, при котором знак идентификатор пишется слева от знака "<< а определяемый артефакт справа;
- постфиксное, когда слева от знака ">>"задается определяемый артефакт, а справа его идентификатор;
- предварительное описание с использованием объявления обозначения (применяется при описании типов данных, функций, хранилищ).

Примечание *Предварительное описание будет представлено ниже. Возможно оно будет вынесено в отдельную категорию.*

```
обозначение = идентификатор "<<" элемент | элемент ">>" идентификатор.
```

Под элементом понимается любой из артефактов языка. Понятия артефакта, выражения и блока будут даны ниже.

```
элемент = артефакт | выражение | блок | обозначение | идентификатор.
```

Примеры:

```
X << 100;
Pi << 3.1415;
10 >> ten;
(a, b):+ >> sum;
x0 << y0 << 0;
```

Примечание. *Следует отметить, что в языке имеется присваивание, которое функционирует по принципу единственного присваивания. Это обусловлено тем, что при введении статической типизации появляются контейнерные артефакты заданной размерности, в которые можно вводить данные, руководствуясь этими размерами. Также имеются аналоги структур (записей), имеющие поля с локальным именованием. В этих случаях тоже удобнее заносить данные внутрь этих конструкций путем единственного присваивания. Это же касается обобщений. Но обозначение все равно остается и будет для таких артефактов использоваться как начальное объявление.*

3.4 Артефакты

К артефактам языка относятся программные объекты, определяемые в языке и несущие заданную семантическую нагрузку. Каждый артефакт характеризуется двойкой:

```
<тип, значение>.
```

В языке используется статическая типизация артефактов, что позволяет сформировать и идентифицировать тип любого из них во время компиляции. Значение определяет величину из множества допустимых значений, определяемых для данного типа.

Примечание. *Наличие строгой статической типизации в целом не отменяет изменчивость типов. Любая маломальская программа требует в явной или неявной форме поддержки динамической типизации данных. Ее последующая реализация в языке планируется через процедурно–параметрический полиморфизм. В первоначальной версии предполагается использование объединений в стиле языка программирования Ada.*

Артефакты могут формироваться как до выполнения программы, так и во время ее выполнения. Артефакт, сформированный до вычислений, является константой заранее предопределенного типа. Существуют различные по структуре категории артефактов, которые можно описать следующим правилом:

```
артефакт_по_структуре = атом | составной.
```

Атомарный артефакт полностью определяется семантикой языка и является неделимым на более мелкие артефакты. Составной артефакт формируется из атомарных и ранее определенных составных артефактов. В ряде случаев при формировании составных артефактов допускается рекурсия.

3.4.1 Типы артефактов

Можно выделить неупорядоченное множество предопределенных типов, задаваемых соответствующими именами. Типы делятся на атомарные и составные. Атомарные типы и области их допустимых значений определяются аксиоматически. Составные типы являются комбинацией атомарных и уже существующих составных артефактов. Они конструируются по заданным правилам.

Следует отметить определенную специфику языка, вытекающую из особенностей модели вычислений. Она заключается в том, что многие артефакты могут использоваться в качестве как данных, так и функций оператора интерпретации. Это проявляется в дуализме артефактов, что ведет к двойственной трактовке типов в зависимости от использования. Поэтому артефакт по типу можно охарактеризовать следующим правилом:

```
артефакт_по_типу = данные | функция | дуальный.
```

Учитывая тип артефакта в зависимости от применения, его можно охарактеризовать следующей конструкцией:

```
<тип-данные : тип-функция, значение>.
```

Двоеточие разделяющее значение типов, показывает их местоположение относительно постфиксного оператора интерпретации. В качестве примера можно привести целые числа. В качестве данных они используются в диапазоне от минимального до максимального целого и имеют тип `int`. В качестве функции они используются как селекторы данных из контейнерных типов. При этом их тип интерпретируется как `func`. Поэтому описание целых чисел выглядит следующим образом:

```
<int:func, MinInt ... MaxInt >.
```

Также следует отметить, что имеющаяся возможность перегрузки имени функции за счет использования идентификации по сигнатуре позволяет связывать с одним обозначением несколько артефактов функционального типа. В таблице 3.1 приведены предопределенные артефакты языка.

Примечание *В текущей версии действительный и символьный тип реализовывать не планируется. Это связано с тем, что первоначально предполагается отработать ключевые конструкции языка, после чего можно переходить к его дальнейшему расширению.*

3.5 Константы

Константы относятся к неделимым атомарным величинам, принимающим конкретные значения, предопределенного для них типа данных. Значение константы принадлежит области ее допустимых значений, задаваемой в зависимости от типа

Таблица 3.1 — Предопределенные артефакты

Название артефакта	**Обозначение типа**	**Организация**
сигнал	`signal:func`	атом
логический	`bool:func`	атом
целый	`int:func`	атом
вектор	`vector:func`	составной
кортеж	`tuple:func`	составной
структура	`struct:none`	составной
обобщение	`union:none`	составной
рой	`swarm:func`	составной
задержка	`none:none`	составной
функция	`none:func`	составной
ошибка	`error:none`	атом
очередь	`queue:func`	составной
типовой	`type:func`	перечислимый
действительный	`float:none`	атом
символьный	`char:none`	атом

одним из следующих способов: диапазоном, диапазоном и точностью, перечислением элементов упорядоченного множества, перечислением элементов неупорядоченного множества (если нет необходимости устанавливать между элементами отношение порядка), функцией. Каждая константа — это одно значение из диапазона, определяемого областью допустимых значений. В языке реализованы следующие виды констант:

- сигнальная константа;
- целочисленные константы
- булевские константы;
- константы ошибок;
- специальные константы.

Тип константы в программе определяется ее внешним видом, задаваемым синтаксическими правилами:

```
константа = сигнальная | целая | логическая.
```

Семантика констант, связана с семантикой их величин, ролью в операторе интерпретации и приводится в описании оператора интерпретации.

3.5.1 Сигнальная константа

Сигнальная константа или просто сигнал имеет предопределенный тип `signal` и может принимать только одно значение `!`, указывающее на произошедшее событие, не связанное с другими типами данных. Кроме фиксации факта возникновения некоторого события сигнал больше не содержит никакой дополнительной информации.

3.5.2 Целая константа

Целая константа имеет предопределенный тип данных `int` и используется для представления данных в формате стандартного машинного слова, длина которого зависит от архитектуры ВС.

Примечание. *В текущей версии языка реализовано представление целых чисел только в десятичной системе счисления. Этого достаточно для проведения первоначальных экспериментов.*

```
целая = [ "+" | "-" ] {/цифра/}.
цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примеры:

```
127
0127
-356
+10
```

Предполагается, что в соответствии с внутренним машинным представлением целочисленные константы располагаются в диапазоне `{MinInt, ..., MaxInt}`.

Например для 64-разрядной архитектуры в дополнительном коде это будет диапазон от -2^{64} до 2^{64} - 1.

3.5.3 Логическая константа

Логическая константа имеет предопределенный тип `bool` и может принимать значения «`true`» («истина») или «`false`» («ложь»). Она задается соответствующими ключевыми словами:

```
логическая = true | false.
```

Для логических констант сохраняется отношение порядка:

```
false < true
```

3.5.4 Константы ошибок

Константы ошибок имеют тип `error` и используются для отображения некорректных ситуаций, возникающих в ходе вычислений. Величины этого типа могут обрабатываться наряду с другими данными или как исключительные ситуации.

Область допустимых значений для констант ошибки задается неупорядоченным множеством, которое в дальнейшем предполагается пополнять. В настоящий момент выделяются следующие ошибки:

- `ERROR` — неидентифицируемая ошибка;
- `REAL_ERROR` — некорректное преобразование действительного числа;
- `INT_ERROR` — некорректное преобразование целого числа;
- `ZERODIVIDE_ERROR` — деление на ноль;
- `INTERPR_ERROR` — ошибка операции интерпретации;
- `BOUND_ERROR` — ошибка выхода за границы диапазона;
- `EMPTY_ERROR` — попытка обращения к опустевшему (завершенному) потоку;
- `BASEFUNC_ERROR` — неправильное использование предопределенной функции;
- `SINGLE_ERROR` — нарушение принципа единственного присваивания;
- `NO_ERROR` — отсутствие ошибки.

Эти имена запрещается использовать в программе в другом контексте.

```
константа_ошибки = ERROR | REAL_ERROR | INT_ERROR |
        ZERODIVIDE_ERROR | INTERPR_ERROR | BOUND_ERROR |
        EMPTY_ERROR | BASEFUNC_ERROR | SINGLE_ERROR | NO_ERROR.
```

Значение `NOERROR` формируется в том случае, если при получении результата вычислений необходимо вместо конкретного значения ошибки вернуть информацию о том, что ошибка отсутствует.

Примечание. *Предполагается, что по сравнению с Пифагором использование ошибок будет значительно переработано в сторону упрощения. Планируется, что каждая из констант ошибки будет являться отдельной функцией - обработчиком ошибки, предоставляющей информацию о типе ошибке и обеспечивающей обращение к системе для корректного завершения программы. На данном этапе не планируется введение системы обработки исключений. Хотя в дальнейшем такое развитие просматривается.*

Возможно, что реализация обработки ошибок будет сделана по аналогии с тем, как это реализовано в языке программирования Go. То есть, вместо выбрасывания исключений, будет формироваться значение, возвращаемое в качестве одного из результатов.

3.5.5 Описание именованных констант

Для любой из представленных выше констант можно ввести обозначение в виде имени. Это позволяет в разных местах программы ссылаться на одну и ту же константу. Описание именованных констант задается следующим синтаксическим правилом:

```
описание_константы = имя_константы "<<" ["const"] константа |
                     ["const"] константа ">>" имя_константы.
```

Ключевое слово const не является обязательным, так как и без его использование описание задается однозначно.

Примеры:

```
ten << 10
const 5 >> five
one << const 1
3.141592 >> pi
ok << true
false >> fail
```

Примечание. *Впоследствии могут появиться константные выражения, которые состоят из других именованных и неименованных констант. Эти выражения будут вычисляться на этапе компиляции и экспортироваться в виде константных значений, доступных через введенное для них обозначение.*

3.6 Описания типов

Язык ориентирован на использование статической системы типов. Предполагается что каждый объявляемый тип должен иметь имя. Существуют предопределенные (базовые) типы и сконструированные типы (типы, определяемые пользователями). Среди предопределенных можно выделить атомарные и составные типы.

Описания позволяют создавать новые именованные типы, определяя их через предопределенные типы или используя уже сконструированные типы. Имя вновь создаваемого типа задается идентификатором, который используется в качестве обозначения. После обозначения следует описание, которое начинается с ключевого слова `type` или эквивалентного по ему по смыслу специального обозначения типа `@`:

```
ОписаниеТипа = ИмяНовогоТипа "<<" ("@" | "type") Тип |
               ("@" | "type") Тип ">>" ИмяНовогоТипа.
```

Примечание. *Двоякое толкование начала описания в данный момент вызвано неопределенностью ответа на вопрос: какое обозначение типа лучше? С одной стороны собака «@» в языке трактуется как обозначение типа. И запись при этом получается компактной. С другой стороны в языках принято (не всегда) ставить ключевое слово type. Решил использовать оба, пока не принято окончательное решение.*

3.6.1 Переименование типа

Основная идея использования переименования типов заключается в создании новых типов на основе уже существующих. При этом новые типы напрямую не наследуют свойства уже созданных типов. В отличие от алиасов (псевдонимов, определяющих дополнительные имена для того же типа) все функции исходного типа над вновь созданным напрямую недоступны. Использовать функции, родительского типа их можно только после явного приведения созданного типа к типу предшественника. Например, допускается использование предопределенных атомарных типов, имеющих имя, для построения новых типов. Например:

```
apple << @ int
```

Ключевым здесь является следующий принцип: если хочется использовать новый именованный тип, то для обработки его нужно сформировать свой набор функций, а не использовать неявно функции родительского типа. В противном случае незачем формировать новый тип, а достаточно использовать уже существующий. Использование общих функций для нескольких типов можно реализовать посредством обобщений.

Примечание. *Это во многом противоречит общепринятой трактовке переименования типа, когда новое имя используется как алиас старого с сохранением всей семантики. Зачастую это бывает полезным для создания нужной ориентации на предметную область минимальными усилиями. Однако можно подумать в дальнейшем над тем, чтобы прямое соответствие сформировать явно за счет дополнительных опций, когда это нужно. Или более экономными средствами сформировать ограниченное соответствие.*

Следует отметить, что использование алиасов тоже возможно. Оно связано с применением обозначением и констант, которые можно применять к любым конструкциям. Поэтому формирования алиаса для любого типа можно осуществить, применив соответствующее обозначение. Например:

```
integer << int
boolean << const bool
```

3.6.2 Конструирование типа

Создание новых типов на основе составных предопределенных типов обеспечивает формирование именованных абстрактных типов данных с требуемыми полями и свойствами. Эти типы могут создаваться на основе любых составных типов и использоваться в различных манипуляциях, допускаемых над величинами, имеющими тип в которых тип является также и значением (вид типа).

Примечание. *Что-то подобное можно заимствовать из Хаскела.*

В данной версии языка предполагается, что будет использоваться только (или в основном) именованная эквивалентность типов. То есть, два типа будут считаться эквивалентными, если они имеют одинаковые имена. Это, на мой взгляд,

обеспечит более строгий контроль типов и позволит проводить формальную верификацию программ в более широком диапазоне. Поэтому в большинстве случаев перед сравнением типов необходимо будет осуществлять явное преобразование к нужному типу, если такое возможно, в соответствии с используемым механизмом преобразования (приведения) типов. Но без структурной эквивалентности в ряде случаев просто не обойтись. Поэтому данный вопрос требует детальной проработки...

Структурная эквивалентность будет использоваться при сопоставлении предопределенных составных типов. В этом случае возможна проверка на идентичность имен предопределенных типов, после чего следует проверка идентичности внутренней структуры. Но при этом предполагается, если это возможно, явное приведение сопоставляемых типов к единому предопределенному составному типу.

3.6.3 Предопределенные атомарные типы

К атомарным относятся следующие предопределенные типы:

- сигнальный тип (signal);
- булевский тип (bool);
- целый тип (int);
- ошибочный тип (error);
- пустой (незаданный) тип (empty).

```
атомарный_тип = сигнальный | булевский | целый |
функциональный | ошибочный | пустой.
сигнальный = "signal".
булевский = "bool".
целый = "int".
пустой = "empty".
```

Сигнальный тип

Сигнальный тип (или просто сигнал) обозначается ключевым словом «`signal`». Он отличается от других атомарных типов тем, что не имеет конкретного значения и в динамике определяется только фактом своего появления. Для того, чтобы показать наличие сигнала, используется константа «`!`». То есть, можно говорить о том, что сигнальный тип определяется только одним значением. Готовность величины сигнального типа определяется самим фактом ее появления.

Появление сигнала качестве результата непосредственно определяет факт срабатывания соответствующего оператора интерпретации. Использование сигналов позволяет, при необходимости, моделировать в функциональных программах явное

управление вычислениями. Они также могут сигнализировать о завершении работы функции, не возвращающей значимый результат.

Любая функция, не имеющая аргументов, может быть запущена только при наличии сигнала в качестве аргумента операции интерпретации. Постоянное присутствие сигнала, определяющее «моментальный» запуск, задается следующим выражением:

```
!:F
```

Формат величины, определяющий внутреннее строение сигнала:

```
<signal:func, { ! }>.
```

По сути данные любого типа содержат сигнал, информирующий об их появлении. Отличие заключается в том, что все прочие типы данных имеют множество допустимых значений мощность которого больше единицы.

Булевский тип

Булевский тип обозначается ключевым словом «`bool`». Данные булевского типа принимают значения из множества булевских констант (`true`, `false`), мощность которого равна двум.

Формат величины, определяющий внутреннее строение булевской величины:

```
<bool:func, {true, false}>.
```

Целый тип

Целый тип обозначается ключевым словом «`int`». Данные целого типа принимают значения из множества целочисленных констант, мощность которого определяется реализацией целых чисел в конкретной компьютерной архитектуре.

Формат величины, определяющий внутреннее строение целочисленной величины:

```
<int:func, {MinInt, ..., MaxInt}>.
```

Ошибочный тип

Ошибочный тип обозначается ключевым словом «`error`». Данные ошибочного типа принимают значения из множества констант ошибок, которое задается путем перечисления видов ошибок, возможных в ходе выполнения функционально-потоковых параллельных программ. Данное множество может изменяться в ходе разработки системы функционально-потокового параллельного программирования. Однако при этом не предполагается, что пользователи могу самостоятельно расширять данное множество, что и позволяет отнести его к классу атомарных типов. Данный тип не может использоваться в качестве функции.

Формат величины, определяющий внутреннее строение величины, задающей ошибку:

`<error:empty, {Множество значений констант, описывающих ошибки}>`.

Примечание. *Следует отметить, что на текущий момент множество атомарных типов не включает ряд типов, которые традиционно присутствуют практически во всех других языках программирования. В частности, отсутствует тип, задающий числа с плавающей точкой. Предполагается, что ядро языка не будет ориентировано на типы данных, связанные с конкретными прикладными вычислениями. Также в нем будут отсутствовать функции, ориентированные на обработку этих типов. Добавление новых типов и функций планируется рассматривать как расширение ядра языка (Праязыка). При этом возможны различные варианты расширений, которые будут определять семейства проблемно–ориентированных дочерних языков (языков — потомков), предназначенных для соответствующих предметных областей и параллельных вычислителей (в основном речь идет об ориентации на различные параллельные вычислительные архитектуры).*

Пустой тип

Понятие пустого типа вводится для обозначения того, что некоторая величина в одном из своих контекстов (данных или функции) не имеет конкретно заданного типа. То есть, она не относится ни к одному из предопределенных типов. Этот тип назначается тогда, когда не имеет смысла придумывать какой-то иной тип для обозначения группы значений. Например, множество знаков, используемых для задания функций сравнения не имеет смысла использовать как некоторой специализированной группы данных в предопределенном контексте:

```
<empty:func, { <, <=, =, !=, >, >=}>
```

Данный тип не допускается использовать для конструирования других типов, так как он не несет требуемого для этого семантического смысла.

3.6.4 Составные типы

Составные типы могут расширяться за счет атомарных типов, составных типов, объектов данных и уточнений составов объектов данных. Они группируют используемые типы в контейнеры или другие структурированные конструкции, обладающие определенными свойствами и поведением, что, в свою очередь, определяет семантику их использования в операторах интерпретации как в роли аргументов, так и в роли функций.

К составным типам языка относятся:

- типовой тип (type);
- обобщение (union);
- любой (выводимый) тип (any);

- функциональный тип (func);
- ссылка (ref).

Данные типы позволяют формировать составные конструкции, которые являются завершенными отдельными элементами, включаемые как в другие составные типы, так и в объекты данных в виде элементов.

В соответствии с концепциями, определяемыми СТМФППВ, в составные типы попадают следующие объекты данных, определяемые как группы:

- соединитель (join), обеспечивающий общую синхронизацию всех поступающих в него данных и формирование сигнала по готовности, когда все его элементы получены;
- рой (swarm), предназначенный для асинхронной упорядоченной группировки, при которой возможно обращение к каждому элементу по его индексу, но поступление элементов не формирует общую готовность роя, а рассматривается для каждого из них независимо;
- поток (stream), определяющий готовность только по первому поступившему элементу, а остальные элементы выстраиваются в очередь в порядке поступления.

Они используются для группировки данных в различные по семантике структуры и обладают специфическим поведением.

Объединение в группы возможны различными способами, что определяется подходами к описанию параллелизма и синхронизации данных. Исходя из этого можно выделить уточняющие параметры, показывающие каким образом элементы групп формируются в единую структурную единицу. Можно выделить следующие варианты объединения элементов в группы:

- вектор (vector);
- массив(?) (array);
- кортеж (tuple);
- структура (struct).

Вектор (векторная группа) обозначается ключевым словом «`vector`. Он обеспечивает группировку данных одного типа, Возможно формирование статических и динамических векторов. Длина статически определяемого вектора задается целочисленным положительным константным выражением вычисляемым во время компиляции. Длина динамически порождаемого вектора вычисляется во время выполнения функции, но до начала его использования. Значение формируемой длины тоже должно являться целым положительным числом.

Примечание. *Принято решение начать нумерацию элементов не с 1, а с 0, как это сейчас принято в большинстве языков программирования. Оно обусловлено тем, что убраны отрицательные значения индексов, используемых в качестве функций, которые убирают соответствующий номеру элемент из вектора. Исчезновение этих функций связано с тем, что при их использовании возвращается результат другого типа, отличающегося от типа элемента.*

Нумерация элементов, размещенных внутри вектора, начинается с нуля. Число N в данном случае определят длину вектора. Описание векторного типа задает для типа его имя, определяемое пользователем. Это описание всегда требует указание длины вектора целочисленным константным выражением и задается с использованием следующего синтаксиса:

Массив(?) (обозначается ключевым словом «`array`) задает многомерную индексированную совокупность элементов одного типа.

Примечание. *На данном этапе разработки создание многомерных массивов не предполагается. Поэтому в дальнейшем они не детально рассматриваются. Однако в комментариях пока присутствуют, чтобы не забыть об их последующем возможном включении.*

Кортеж (tuple), предназначен для группирования неоднородных данных. По сути он похож на вектор, но может содержать разнотипные элементы. Обращение к элементам кортежа осуществляется по номеру поля (поля, как и в векторе, нумеруются, начиная с нуля).

Структура (struct) аналогично кортежу обеспечивает группировку разнотипных данных по аналогии со структурными типами различных языков программирования. Доступ к элементам осуществляется по именам полей, связанных с каждым элементом структуры (помимо этого считается, что поля упорядочены в порядке их описания, что позволяет осуществлять доступ с использованием индексации как и для кортежей?).

Исходя из этого можно рассматривать различные варианты специализации для групп:

- соединитель–вектор;
- соединитель–кортеж;
- соединитель–структура;
- рой–вектор;
- рой–кортеж;
- рой–структура.

Типовой тип

Типовой тип (вид типа) обозначается ключевым словом type или символом "@". Данные этого типа принимают значения из множества имен предопределенных типов и типов, созданных пользователем и используемых в текущей программе. По

сути это некоторый перечислимый тип, значения которого расширяются по мере порождения пользователем новых именованных типов, определяемых через описания типов. Это расширение и позволяет относить данный тип к составным.

```
типовой = "type" | "@".
```

Примечание. *Предполагается, что на текущем этапе данный тип реализовываться не будет информация о нем оставлена, чтобы не забыть обдумать его в дальнейших разработках.*

Скорее всего этот тип может стать интересным, если в язык добавить вывод типов.

Формат величины, определяющий внутреннее представление типового типа:

```
<data_type:func, {Множестов имен предопределенных и сконструированных типов}>.
```

Типы «Соединитель–группа»

Соединитель предназначен для общего объединения элементов в единую композицию, которая после поступления всех данных рассматривается как единый элемент. Его окончательное формирование сопровождается выдачей сигнала, информирующего о готовности данных. В соответствии с вариантами специализаций для соединителя возможны следующие группировки элементов:

- соединитель–вектор;
- соединитель–кортеж;
- соединитель–структура;

Тип **соединитель–вектор** обеспечивает использования вектора, готовность которого к выполнению определяется при поступлении всех его элементов. Возможно формирование статических и динамических соединителей–векторов. Длина статически определяемого соединителя–вектора задается константным выражением, значение которого должно быть положительной целочисленной величиной вычисляемой во время компиляции. Длина динамически порождаемого соединителя–вектора вычисляется во время выполнения программы, но до начала его использования. Ее значение тоже должно являться целым числом.

Описание типа соединитель–вектор требует указание длины вектора целочисленным выражением и задается с использованием следующего синтаксиса:

```
соединитель-вектор = ["vector"] имя_типа "(" длина ")".
длина = ЦелочисленноеКонстантноеВыражение | ЦелочисленноеВыражение.
```

Примеры описания векторных типов:

```
A << @ int(100)             // Целочисленный соединитель-вектор типа A
B << @ vector bool(30)      // Булевский соединитель-вектор типа B
```

Ключевое слово «vector» в описании типа является необязательным. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Элементы соединителя–вектора задаются в виде списка однотипных значений, заключенных в круглые скобки с префиксом, определяющим вектор:

```
vector(элемент[0], элемент[1], ..., элемент[N-1])
```

Значения векторов записываются в скобках аналогично тому, как записываются и значения кортежей. Поэтому обычная запись:

```
(1, 3, 2, 10, 6)
```

является кортежом длиной, равной 5, несмотря на то, что все элементы имеют одинаковыйх тип. Для описания векторов необходимо явно задать тип или сделать приведение кортежа к векторному типу. Например:

```
vector(1, 3, 2, 10, 6)
(1, 3, 2, 10, 6):vector
```

Тип **соединитель–кортеж** предназначен для группирования неоднородных данных. По сути он похож на вектор, но может содержать разнотипные элементы. Обращение к элементам кортежа осуществляется по номеру поля (поля, как и в векторе, нумеруются, начиная с нуля). Для задания соединителей–кортежей используется следующий синтаксис:

```
соединитель-кортеж = ["tuple"] "(" ИмяТипа ["*" Множитель]
{ "," ИмяТипа } ["*" Множитель] ")".
Множитель = Целое.
```

Множитель позволяет задать коэффициент повторения для типа, который повторяется несколько раз подряд. Это целое положительное число.

Примеры задания типов кортежей:

```
C << @ (int)
B << @ (int, bool, signal)
D << @ (int*5, bool*3, signal*7)
```

Ключевое слово «**tuple**» не является обязательным в описании кортежа. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации. Возможно явное преобразование структур в кортеж, что обуславливается эквивалентностью представления типов данных внутри структур и кортежей. Помимо этого возможно явное приведение к типу кортеж векторов. Обратное приведение допустимо, если все элементы кортежа имеют один тип. Допускается также явное приведение кортежей к именованным структурам, типы элементов которых попарно совпадают с типами элементов кортежа.

Тип *соединитель–структура*, как и кортеж, обеспечивает группировку разнотипных данных по аналогии со структурными типами различных языков программирования. Соединитель–структура состоит из полей, каждое из которых имеет имя и тип. Описание структуры имеет следующий синтаксис:

```
Структура = ["struct"] "(" ПолеСтруктуры
{ "," ПолеСтруктуры } ")".
ПолеСтруктуры = ИмяПоля "@" ИмяТипа
| ИмяПоля { "&" ИмяПоля }  "@" ИмяТипа.
```

Примеры структурных типов:

```
Triangle << @ (a@int, b @ int, c @int)
Triangle2 << @ (a & b @ int, c @int)
Rectangle << @ (x&y@int)
```

Ключевое слово `struct` не является обязательным в описании структуры. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Тип **соединитель–массив**.

Примечание. *Следует отметить, что одной из идей является использование многомерных массивов, задаваемых с использованием ключевого слова* `array`. *Это расширяет параллелизм на многомерные конструкции. Возможное их описание может быть представлено следующим синтаксисом:*

```
Массив = ИмяТипа "(" Размерность ")".
Размерность = (ЦелочисленноеКонстантноеВыражение
| ЦелочисленноеВыражение)
{ "," (ЦелочисленноеКонстантноеВыражение
    | ЦелочисленноеВыражение) }.
```

Примеры массивов:

```
AA << @ int(100, 100)
BB << @ bool(30, 40)
```

Однако пока мне непонятно, каким образом лучше реализовать массивы. Прямое решение не выглядит достаточно эффективным из-за проблем с использованием многоразмерных индексных выражений. В дальнейшем, как вариант, предполагается рассмотреть реализацию массивов в виде расширения векторов. То есть в виде некоторой оболочки, которая добавляет индексы к вектору. Тогда для выбора любого элемента массива планируется преобразование его индексов к индексу вектора путем специальной операции index. Например: `A:index^(i,j,k)`. Этот вариант видится мне более предпочтительным. Не смотря на то, что вектор уже не будет частным случаем массива, данный вариант позволяет рассматривать приведение массива к вектору и вектора к массиву...

На данном этапе реализацию многомерных массивов реализовывать не планируется. Описание оставлено, чтобы не забыть проанализировать возможности реализации в будущем.

Типы «Рой–группа»

Рой (роевой тип) используется для описание независимых данных, над которыми возможно выполнение массовых параллельных операций. Обозначается ключевым словом **swarm**. Все элементы роя имеют один тип, выполнение операций с элементами роя может осуществлять по мере их поступления независимо друг от друга.

- рой–вектор;
- рой–кортеж;
- рой–структура;

Тип **рой–вектор** обеспечивает асинхронное использования элементов вектора по мере их поступления. Возможно формирование статических и динамических роев–векторов. Длина статически определяемого соединителя–вектора задается константным выражением, значение которого должно быть положительной целочисленной величиной вычисляемой во время компиляции. Длина динамически порождаемого роя–вектора вычисляется во время выполнения программы, но до начала его использования. Ее значение тоже должно являться целым числом.

Описание типа рой–вектор его требует указание длины вектора целочисленным выражением и задается с использованием следующего синтаксиса:

```
рой-вектор = ["vector"] имя_типа "[" длина "]".
длина = ЦелочисленноеКонстантноеВыражение | ЦелочисленноеВыражение.
```

Примеры описания векторных типов:

```
A << @ int[100]             // Целочисленный рой-вектор типа A
B << @ vector bool[30]      // Булевский рой-вектор типа B
```

Ключевое слово «`swarm`» в описании типа является необязательным. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Элементы роя–вектора задаются в виде списка однотипных значений, заключенных в прямоугольные скобки с префиксом, определяющим вектор:

```
vector[ элемент[0], элемент[1], ..., элемент[N-1] ]
```

Значения векторов записываются в скобках аналогично тому, как записываются и значения кортежей. Поэтому обычная запись:

```
[1, 3, 2, 10, 6]
```

является кортежом длиной, равной 5, несмотря на то, что все элементы имеют одинаковыйх тип. Для описания векторов необходимо явно задать тип или сделать приведение кортежа к векторному типу. Например:

```
vector[1, 3, 2, 10, 6]
[1, 3, 2, 10, 6]:vector
```

Тип **рой–кортеж** предназначен для группирования неоднородных данных. По сути он похож на вектор, но может содержать разнотипные элементы. Обращение к элементам кортежа осуществляется по номеру поля (поля, как и в векторе, нумеруются, начиная с нуля). Для задания роев–кортежей используется следующий синтаксис:

```
соединитель-кортеж = ["vector"] "[" ИмяТипа ["*" Множитель]
    { "," ИмяТипа } ["*" Множитель] "]".
Множитель = Целое.
```

Множитель позволяет задать коэффициент повторения для типа, который повторяется несколько раз подряд. Это целое положительное число.

Примеры задания типов кортежей:

```
C << @ [int]
B << @ [int, bool, signal]
D << @ [int*5, bool*3, signal*7]
```

Ключевое слово «**tuple**» не является обязательным в описании кортежа. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации. Возможно явное преобразование структур в кортеж, что обуславливается эквивалентностью представления типов данных внутри структур и кортежей. Помимо этого возможно явное приведение к типу кортеж векторов. Обратное приведение допустимо, если все элементы кортежа имеют один тип. Допускается также явное приведение кортежей к именованным структурам, типы элементов которых попарно совпадают с типами элементов кортежа.

Тип *рой–структура*, как и кортеж, обеспечивает группировку разнотипных данных по аналогии со структурными типами различных языков программирования. Соединитель–структура состоит из полей, каждое из которых имеет имя и тип. Описание структуры имеет следующий синтаксис:

```
Структура = ["struct"] "(" ПолеСтруктуры
{ "," ПолеСтруктуры } ")".
ПолеСтруктуры = ИмяПоля "@" ИмяТипа
| ИмяПоля { "&" ИмяПоля }  "@" ИмяТипа.
```

Примеры структурных типов:

```
Triangle << @ (a@int, b @ int, c @int)
Triangle2 << @ (a & b @ int, c @int)
Rectangle << @ (x&y@int)
```

Ключевое слово `struct` не является обязательным в описании структуры. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Тип «Поток»

Описание потока в языке программирования Smile определяется следующим синтаксисом:

```
поток = имя_типа_элементов "<[]".
```

Тип Обобщение

Тип Обобщение (обобщающий тип) во многом аналогичен по организации и использованию обобщениям, используемым в других языках. Основной его задачей является объединение воедино взаимоисключающих артефактов. Существуют различные подходы к организации обобщений, включая методы, поддерживающие полиморфизм. В языке предполагается использование процедурно-параметрических обобщений, обеспечивающих более гибкую поддержку эволюционного расширения программ по сравнению с другими подходами. Правила, определяющие синтаксис обобщений имеют следующий вид:

```
Обобщение = ["union"] "{" ПолеОбобщения
    { "," ПолеОбобщения } "}".
ПолеОбобщения = ИмяТипа { "," ИмяТипа }
| ИмяПризнака "@" ИмяТипа
| ИмяПризнака { "&" ИмяПризнака } "@" ИмяТипа.
```

Ключевое слово **union** не является обязательным в описании обобщения. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Примеры описания обобщений:

```
Figure1 << @ {Triangle, Rectangle}
Figure2 << @ {
    trian@Triangle,
    rect@Rectangle,
    rhomb@Rectangle
}
WeekDay << @ {Sun & Mon & Tue & Wen & Thu & Fri & Sat @signal}
```

Примечание. *Пока вопрос с обобщением и их использованием рассматривается на уровне мономорфизма. То есть, предполагается явный анализ признаков обобщения, аналогичный тому, как это делается с объединениями в Си и вариантными записями в Паскале. Применение процедурно-параметрического полиморфизма или других вариантов планируется только в следующих версиях.*

Следует также отметить наличие в языке глобального обобщения, которое может объединять в единую конструкцию любые именованные типы данных. В этом случае обобщение обеспечивает поддержку в языке динамической типизации и по сути

является аналогом вариантных данных различных языков. Допускает приведение произвольных значений к данному типу с последующим возможным анализом имени типа полученного значения и соответствующим его выделением для выполнения необходимых операций.

Примечание. *С другой стороны этот тип может рассматриваться как процедурно-параметрическое обобщение общего вида, формируемое автоматически сборкой имен типов в качестве признаков. То есть, можно будет создавать обработчики обобщений с использованием в качестве обобщающих аргументов тип* `any`.

Пока я обозначил этот тип. Более конкретный его анализ, включая полноту операций, планируется провести позднее. Также пока непонятно, стоит ли вообще вводить этот тип. Возможно, что он может оказаться полезным, когда будет рассматриваться добавление вывода типов с неопределенными атрибутами во время компиляции.

Глобальные обобщения относятся к типу `union`. Формируемые именованные параметрические обобщения имеют тип, соответствующий заданным именам типов.

Любой тип

Любой тип обозначается ключевым словом `any`. По сути это не обозначение конкретного типа, а понятие, показывающее, что в данном месте программы может находиться любой именованный тип, описанный в программе. В ходе компиляции программы вместо `any` осуществляется подстановка конкретного типа, выводимого из контекста. То есть, данное понятие используется системой вывода типов. В основном `any` используется в прототипах функций для описания типов, которые могут принимать различные значения, включая и пустой тип. Основная задача данного понятия заключается в обеспечении системы вывода типов. Также используется при описания прототипов в документации для того, чтобы показать наличие на данном месте любого конкретного типа.

Тип Рой

Тип Рой (роевой тип) используется для описание независимых данных, над которыми возможно выполнение массовых параллельных операций. Обозначается ключевым словом **swarm**. Все элементы роя имеют один тип, а функция, осуществляющая их обработку, может одновременно выполняться над каждым элементом. Результатом является также рой, размерность которого равна размерности роя аргументов. Синтаксические правила, определяющие данный тип, имеют следующий вид:

```
Рой = ["swarm"] ИмяТипа "[" Целое "]".
```

Ключевое слово **swarm** не является обязательным в описании роя. Оно также может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Пример описания типа:

```
R << @ int[100]
```

Рой обеспечивает группировку данных одного типа в вектор, готовность которого к выполнению определяется при поступлении любого его элементов. В отличие от вектора оператор интерпретации начинает реагировать на поступление в рой каждого элемента вместо ожидания момента, когда данные полностью сформируются. Это может обеспечить запуск и частичное выполнение функции, обрабатывающий рой до того момента, когда будут готов все его данные.

Возможно формирование роев фиксированной и переменной размерности, каждый из которых обладает своими свойствами и может обрабатываться своим множеством функций. Множества функций для обработки разных видов роев пересекаются, но не перекрываются. Элементы роя задаются в виде списка значений, заключенных в квадратные скобки:

```
[элемент0, элемент2, ..., элементN-1]
```

Нумерация элементов, размещенных внутри роя, начинается с нуля. Число N в данном случае определят размер роя. Размер роя может быть задан константным выражением при описании его типа в том случае, если объявляется рой с фиксированной размерностью.

Примечание. *Как и для вектора пока предлагается только одномерное решение. Хотя есть соблазн разобраться и с реализацией многомерных роев. Но пока данный вопрос остается нерешенным.*

Тип Очередь

Тип Очередь является альтернативой асинхронному списку языка программирования Пифагор. Он используется для обработки данных поступающих асинхронно в произвольные промежутки времени. Количество поступающих данных при этом неизвестно, поэтому завершение обработки возможно только по признаку конца поступления данных. Очередь готова к обработке при наличии в ней хотя бы одного элемента. Тип всех элементов осереди одинаков. Синтаксические правила, определяющие очередь:

```
очередь = ["queue"] ИмяТипа "{}".
```

Ключевое слово **queue** не является обязательным в описании очереди. Оно может использоваться при проверке типа, а также в качестве имени функции при выполнении оператора интерпретации.

Пример описания типа Очередь:

```
A << @ int{}
B << @ queue bool{}
```

Функциональный тип

Тип Функция (или функциональный тип) обозначается ключевым словом **func**. Позволяет обозначить сигнатуру функции, определяя тип аргумента и тип результата. В целом определение функционального типа отличается от общепринятых во многих других языках программирования только тем, что функция имеет только один аргумент и возвращает только один результат. Синтаксические правила, определяющие описание функционального типа:

```
ФункциональныйТип = ("func" | "\") Аргумент "->" Результат
Аргумент = ИмяТипа | КортежТипов
Результат = ИмяТипа | КортежТипов
КортежТипов = "(" ИмяТипа "," { ИмяТипа } ")"
```

Примеры описаний:

```
F << @ func int -> int
F2 << @ func (bool, int, int) -> (int, bool)
F3 << @\int -> int
```

Примечание. *Принято решение о возможном использовании знака «\» в качестве альтернативы ключевому слову func. Это достаточно субъективное решение. Оно навеяно описанием лямбда функций в Хаскеле. Но ничто не мешает его использованию и здесь. Тем более, что представленный тип описания, как и описание самих функций с этим знаком синтаксически ничему не противоречат. Эти же альтернативные обозначения можно будет использовать и при задании лямда функций, если они появятся в языке (когда–то появятся...).*

С каждым функциональным типом связываются тип аргумента, обрабатываемого функцией, и тип формируемого результата, образуя сигнатуру функции. Сигнатура определяет принадлежность всех функций к одному типу. При этом любая функция, соответствующая данной сигнатуре, может восприниматься как некоторая величина, областью допустимых значений (ОДЗ) которой является совокупность ОДЗ ее аргумента и результата, что как раз и задается функциональным типом.

Примечание. *Возможны в перспективе варианты, когда функция возвращает вектор, массив, рой или очередь. Но пока так глубоко копать не буду...*

Интерес также представляет задание структуры в качестве типа. Но в данной ситуации предполагается, что использование кортежа обеспечивает структурную эквивалентность и позволяет в дальнейшем подставлять любые структурно эквивалентные типы...

Нужно по тексту четко отделить прототип от сигнатуры. Пока это не сделано.

Следует отметить, что наряду с сигнатурой используется понятие прототипа функции, которое включает имя функции, тип аргумента и тип результата. Данное

понятие полезно при описании перегрузки функций, заключающейся в том, что допускаются функции, имеющие одинаковые имена, но разные сигнатуры. При этом необходимо, чтобы у сигнатур отличались типы аргументов. Прототипы функций с одинаковыми именами и типами аргументов в языке недопустимы.

Тип Ссылка

Тип Ссылка (или ссылочный тип) обеспечивает поддержку указателей на различные хранилища определенного типа, что позволяет передавать значения между функциями без их копирования. Основное назначение заключается в дополнительном контроле типов в ходе передач. Синтаксические правила, определяющие описание ссылочного типа:

```
Ссылка = ИменованнаяСсылка
| ССылкаНаВектор
| ССылкаНаРой
| ССылкаНаПоток.
ИменованнаяСсылка = "&" ИмяТипа.
```

Ссылки обеспечивают доступ к данным, имеющим соответствующий именованный тип. В зависимости от того, каким образом сформированы эти данные, через ссылку может осуществляться как их чтение, так и запись. В последнем случае должен соблюдаться принцип единственного присваивания, если установлена неизменяемость данных.

При взаимодействии с контейнерными данным зачастую вместо именованных типов удобнее использовать данные, имеющие отличающуюся размерность, это затрагивает векторы, массивы, рои. Для описания в этих случаях предлагаются ссылки, в которых размерность не устанавливается:

```
ССылкаНаВектор = "&" ИмяТипа "(" ")".
ССылкаНаРой = "&" ИмяТипа "[" "]".
ССылкаНаПоток = "&" ИмяТипа "{" "}".
```

Примечание. *Тоже пока вызывает много вопросов. Требует проработки... Возникла мысль, что данный тип может и не нужен. Вместо него достаточно использовать соответствующие ссылки как обозначения, присоединяемые к величинам и хранилищам. Нужно думать...*

В частности следует отметить, что со ссылками можно связать дополнительный артефакт, обеспечивающий независимое сопровождение при обработке ряда составных типов. Например, очередей и роев. В этом случае формируются дополнительные конструкции, обеспечивающие для каждой из ссылок независимый обход данных. Поэтому нужно думать...

В перспективе возможно добавление ссылки на массив:

```
ССылкаНаМассив = "&" ИмяТипа "(" РазмерностьМассива ")". (???)
```

3.6.5 Преобразование (приведение) величин разных типов

Преобразование типов обеспечивает определенную гибкость в манипуляции абстракциями. Можно выделить статическое и динамическое преобразование типов. При статическом преобразовании типов этим процессом занимается компилятор. То есть, все возможные ошибки определяются на этапе компиляции. Динамическое приведение типов осуществляется во время выполнения программы и связано в основном с преобразованиями обобщений посредством функций преобразования типов, которые вводятся в язык для повышения гибкости. Описание этих функций будет сделано при рассмотрении оператора интерпретации. Следует также отметить, что преобразования осуществляются не с самими типами а над величинами, имеющими некоторый тип, которые преобразуются к величинам другого типа.

Статические преобразования типов во время компиляции

Возможны следующие статические преобразования типов величин:

1. Тип любой величины может быть преобразован к сигнальному типу. При этом значение исходной величины теряется. По сути сигнальный тип является своеобразным обобщением всех других типов, так как он несет только информацию (сигнал) о готовности данных. Следует также отметить, что получение любого другого типа из сигнального невозможно.

2. Именованный тип любой величины может быть преобразован к типу union. В этом случае формируется структура, определяющая любой тип, которая хранит имя преобразуемого типа, а также сохраняет его значения. Дальнейшие манипуляции с полученной величиной, включая преобразование ее типа, при отсутствии системы вывода типов, могут проводиться только во время выполнения программы.

3. Именованный тип, сформированный определением типа в описании type на основе другого именованного типа, может быть явно преобразован к своему родительскому типу на этапе компиляции. При этом значение программного артефакта не меняется. Данное приведение позволяет выполнять над имеющимся значением все операции над родительским типом. В принципе, используя механизм трансформации, можно изменить поведение преобразования по умолчанию. При этом может измениться значение величины во время выполнения. Однако тип преобразованного значения все равно определяется во время компиляции.

4. Именованный родительский тип, используемый при определении нового типа в описании type, может быть явно преобразован к этому новому типу на этапе компиляции. При этом значение существующей величины не изменяется. Над преобразованной величиной возможны только функции, определенные над этим новым типом. Здесь ситуация обратная предыдущему случаю.

То есть, наличию сформированного инкапсулированного набора можно использовать только после явных приведений типа. Также возможно создание функции трансформации, изменяющей поведение функции используемой по умолчанию.

5. Формируемые в программе величины, имеющие структурный тип, могут быть явно преобразованы к кортежам на этапе компиляции. Это преобразование полезно при использовании различных структур в качестве аргументов функций без дополнительный явных приведений.

Примечание. *Неявные преобразования структур, векторов и т.д. к кортежам и обратно решил запретить*

Представленные преобразования могут осуществляться только явно. Явные преобразования предполагают непосредственное использование функций приведения типов, обеспечивающих получение соответствующих новых типов. Например:

```
apple << @int
6: apple                    @int => @apple
(3,4):intPair               @(int, int) => @intPair
(1,2,3,4,5):vector:signal   @(int, int, int, int, int) => @int(5) => @signal
```

Неявные преобразования запрещены, так как могут привести к коллизиям и неопределенному поведению. Например:

```
:+ << func x@intPair->@int {x:tuple:-:return}
// непонятно, какую функцию сложения использовать
(3,4):+     => 7
// Поэтому:
(3,4):intPair:+ => -1
```

Примечание

Обозначение ":+"задает перегрузку знака "+"в качестве функции.

Возможно ситуация, когда имя функции перегружено, а ее уникальность определяется сигнатурой. В этой ситуации возможны функции с разной сигнатурой, аргументы которых имеют типы, созданные от одного родителя. Тогда невозможно определить, какую из функций подставить. В этой ситуации требуется явное приведение типа. Например:

```
apple << @ int;
pear  << @ int;
f1 << func x@apple->signal {...}
f1 << func x@pear->signal {...}
5:pear:f1  - явное приведение для выбора нужной функции
```

Преобразование между атомарными константами

Зачастую при использовании констант удобнее использовать неявные преобразования для восприятия написанного кода. В частности во многих языках программирования целочисленный ноль (0) в зависимости от контекста может восприниматься как действительное число (0.0). Аналогичные неявные преобразования значений констант предполагается использовать и в разрабатываемом языке в тех случаях, когда контекст константы очевиден. В частности на данный момент просматриваются следующие неявные преобразования:

Для булевских величин вместо **true** и **false** допускается неявно использовать целочисленные константы **0** и **1** соответственно. Эту возможность предполагается использовать для разработки функций, реализующих булевские функции, которые впоследствии планируется транслировать в топологию ПЛИС. Удобнее в этом случае манипулировать числами, отображающими булевские значения. Преобразование осуществляется на этапе компиляции.

3.7 Данные (величины)

Данные формируются в ходе вычислений. Они отображаются на память компьютера. Если в языке Пифагор для их представления достаточно было использовать результаты выполнения функций, увязанные с обозначениями, то применение статической типизации наряду с этим подходом позволяет, используя принцип единственного присваивания, создавать незаполненные хранилища требуемого типа, которые в ходе вычислений могут наполняться. Идея введения хранилищ непосредственно связана с использование статической типизации. Можно изначально сформировать ресурсы, в которые данные записываются не из одной выполненной функции, а независмо и различных функций параллельной программы. С одной стороны это противоречит принципам функционального программирования, но вполне соответствует управлению вычислениями по готовности данных. Несмотря на усложнение семантики языка, данный подход позволяет повысить эффективность разрабатываемого кода за счет прямого доступа к заранее сформированным хранилищам вместо их формирования и заполнения во время выполнения программы с использованием программоформирующих операторов. При этом ранее используемый функциональный подход остается, что позволяет писать программы и в чисто функциональном стиле.

При описании задается имя хранилища и его тип:

```
Хранилище = ИмяХранилища "@" ( ИмяТипа | СоставнойТип | Ссылка ).
```

По сути это декларативное описание, предваряющее использование данного имени.

Примечание. *Следует уточнить и описать подробнее.*

Есть подозрение, что ссылку как тип можно убрать, сделав ее явным обозначением путем соответствующего именования

В принципе хранилища могут быть как внутри функции, так и вне их. Последнее позволяет заполнять хранилища из разных функций, минуя параметры и создавая побочные эффекты. Пока планируется реализовать использование хранилищ только внутри функций.

Примеры:

```
x@int
y@intPair
point@(int, int)
&pointRef@point // ссылка на хранилище point (???) -- пока непонятно...
vectorRef @&bool() ???
rectangle@([a,b]@int)
```

3.8 Операторы

Операторы языка определяются в соответствии со статически типизированной функционально-потоковой моделью параллельных вычислений. Они обеспечивают формирование каркаса функции. Выделяются операторы группировки и операторы интерпретации.

Операторы группировки обеспечивают формирование значений составных типов. К ним относятся:

- оператор, обеспечивающий создание кортежей (tuple);
- оператор формирования роя (swarm);
- оператор загрузки в очередь (queue);
- оператор задержки (delay).

Примечание. *Наряду с этими основными операторами нужно обратить внимание, что статическая типизация позволяет создавать составные типы с фиксированной структурой, доступ к элементам которых может обеспечиваться записью в отдельные поля с соблюдением принципа единственного присваивания.*

Необходимо не забыть и описать соответствующие группы операторов для выше определенных типов данных с фиксированной структурой. Это описание необходимо добавить в раздел, посвященный описанию семантики.

Оператор интерпретации обеспечивает формирование функциональных преобразований. Один из его входных параметров является функцией, а другой определяет аргумент, обрабатываемый этой функцией. Используются префиксная и постфиксная формы оператора интерпретации. В префиксной форме оператор обозначается знаком `^`. В этом случае функция `F(x)` будет записана как `F^x`. Оператор интерпретации в постфиксной форме задается двоеточием (`:`), что позволяет записать приведенную функцию в виде `x:F`.

Данный оператор по сути обеспечивает поддержку семантики всех функциональных преобразований, которая представлена в следующем разделе. По сути это единственная явно задаваемая функция языка. Все функциональные преобразования определяются через этот оператор.

Существуют также групповой оператор интерпретации, который предназначен для описания параллельных вычислений. Его синтаксическое отличие заключается в написании каждого из знаков интерпретации дважды: `^^` или `::`. Семантика данного оператора также представлена в следующем разделе.

3.9 Функция

Функция – составной артефакт, конструируемый специальным образом. Она задается определением, начинающимся с ключевого слова **func** или знака \. Функция состоит из заголовка и тела. В заголовке указывается идентификатор аргумента, обеспечивающего передачу в тело функции необходимых данных и его тип, а также тип результата, возвращаемого из функции. В теле описывается алгоритм обработки аргумента с применением соответствующих элементов языка. Доступ к исходным данным осуществляется только через аргумент. Тело функции состоит из элементов, заключенных в фигурные скобки и разделяемых между собой символом ";".

В ходе выполнения функции обычно формируется результат, который возвращается после применения к нем функции возврата, обозначаемой зарезервированным идентификатором **return**:

```
результат:return
или return^результат
```

Помимо этого ключевое слово **return** может использоваться в качестве альтернативного варианта точки возврата, когда оно задает не функцию, а обозначение возвращаемого значения:

```
результат >> return
или return << результат
```

Этот вариант предназначен для случаев, когда из функции необходимо возвратить задержку.

Примечание. *Наличие подобного свойства еще нужно обдумать... Пока же о реализации говорить рано.*

Возвращаемый результат может быть любым допустимым значением, полученным в ходе вычислений. Возврат результата может осуществляться до завершения выполнения всех операций в теле функции, которая продолжает существования до завершения всех внутренних операций. В этом случае в теле функции будет наблюдаться побочный эффект, который может быть связан либо с какими-то дополнительными фоновыми действиями, либо с ошибкой в ее реализации. Однако повторного возврата, в соответствии с принципом единственного присваивания,

произойти не может. Избавление от побочных эффектов, если они являются лишними, осуществляется путем анализа и оптимизации тела функции соответствующими методами анализа.

Если в качестве результата задается рой, то возможен асинхронный (не одновременный) возврат его независимых элементов. Также в качестве возвращаемого значения может выступать сигнал.

```
Функция = ("func" | "\") [ Аргумент ]
    ["->" ТипРезультата] [ТелоФункции].
Аргумент = ИмяАргумента "@" ИмяТипа
    | ИмяАргумента "@" "(" ИмяТипа { "," ИмяТипа } ")"
    | "(" ИмяПоля "@" ИмяТипа { "," ИмяПоля "@" ИмяТипа } ")".
ТелоФункции = [ "{" [ Элемент {";" Элемент }] "}" ].
```

Примечание. *Пока не проработаны и не включены варианты, связанные с использованием в качестве прямого аргумента очередей и других артефактов. Но скорее всего из добавление пойдет как именованных типов*

При отсутствии в определении функции аргумента предполагается, что он имеет сигнальный тип, использование которого в теле функции не имеет смысла. Однако такой аргумент всегда присутствуе при вызове функции, определяя момент ее запуска. В этом случае сигнал может задаваться константой, указывая на немедленный запуск функции. Или же он может являться вычисляемым значением, что позволяет отложить запуск функции без параметров на некоторое время.

Тип возвращаемого значения также может не указываться. В этом случае предполагается что функция возвращает сигнал. Также возврат сигнального значения может быть задан явно.

Отсутствие у функции тела интерпретируется как ее **предварительное объявление**. Оно полезно, когда функция еще окончательно не реализована, но знание ее сигнатуры необходимо в других функциях, например, при наличии рекурсивных вызовов. Следует отметить, что окончательное определение функции, включающее ее тело должно быть сформировано до момента сборки программы.

Под программой понимается функция, которая связана с совокупность полностью определенных и взаимосвязанных функций, обеспечивающих решение поставленной задачи.

Примечание. *Наверное стоит добавить пример функции. Вычисление площади прямоугольника? Более сложные примеры с условиями и прочим наверное лучше добавить при описании семантики, когда будут введены основные конструкции, используемые в примерах.*

3.9.1 Перегрузка имен функций

В языке поддерживается статический полиморфизм, реализуемый за счет перегрузки имен функций. Это предполагает идентификацию функций по уникальности

сигнатуры, состоящей из имени функции, типа аргумента и типа результата. В целом для идентификации функции используются только ее имя и тип аргумента, так как запрещается использовать функции у которых эти два параметра совпадают.

3.9.2 Определение спецзнаков в качестве имен функций

Специальные знаки могут использоваться в качестве имен функций за счет их специального обозначения, имеющего следующий формат:

```
СпециальноеИмя = ":" Спецзнак.
```

Допускается также перегрузка функций, обозначенных специальными именами.
Примечание. *Тоже стоит добавить пример.*
То есть в языке предполагается допускать пользовательскую перегрузку спецсимволов, не противоречащую их начальному определению на уровне языка. Использование двоеточия в качестве префикса возможно, так как оператор интерпретации перегружать запрещено по определению.
Данную возможность предполагается ввести позднее.

3.9.3 Базовые (предопределенные) функции

Базовые функции задаются их именами, являющимися зарезервированными словами и спецсимволами. Большинство этих функций могут выполняться во время компиляции константных выражений. Данные функции определяют основные операции, обеспечивающие, наряду с программо-формирующими операторами формирование структуры программы и выполнение предопределенных вычислений.

3.10 Блок

Блок - это объединение элементов внутри тела функции, служит для логического соединения группы операторов выполняющих законченное действие, а также для локализации обозначений. Он начинается с ключевого слова **block**, за которым следует тело блока, аналогичное телу функции. Отличие тела блока заключается в том, что выход из него осуществляется по обозначению результата зарезервированным идентификатором **break**, с которым связывается значение, возвращаемое из блока. Данное ключевое слова (как и **return**) может выступать в качестве функции или обозначения.

Использование **break** в качестве функции:

```
результат:break или break^результат
```

Использование **break** в качестве обозначения:

```
результат >> break или break << результат
```

Тип значения, возвращаемого из блока, должен быть известен компилятору и определяется из анализа элементов, входящих в блок или явно задается программистом.

```
Блок = "block" "{" [ Элемент {";" Элемент }] "}".
```

3.11 Выражение

Выражение - это терм или цепочка термов, связанных между собой операторами интерпретации. Под термом понимается артефакт, блок или имя ранее обозначенного элемента. Наличие операции интерпретации позволяет трактовать два ее операнда как функцию и аргумент. Существуют префиксная и постфиксная формы записи оператора интерпретации, отличающиеся друг от друга только порядком следования аргумента и функции. Префиксный оператор интерпретации задается стрелкой вверх "^"или двойной стрелкой ^^, слева от которой стоит терм, выступающий в роли функции, а справа - аргумент:

```
F^X
Q^^Y
```

При постфиксной записи эти же выражения будут выглядеть следующим образом:

```
X:F
Y::Q
```

В том случае, если оператор интерпретации возвращает ошибку, выполнение текущей функции прекращается. Ошибка порождает соответствующие системные сообщения и инициирует прерывание выполнения программы.

```
выражение = терм {( "^" выражение | ":" терм )}.
терм = артефакт | блок | идентификатор.
```

Приведенный синтаксис выражения показывает, что префиксный оператор интерпретации выполняется справа налево, а постфиксный слева направо. Изменение приоритетов можно осуществить использованием блоков, а также фигурных, квадратных или круглых скобок, являющихся операторами группировки в составные конструкции, и, следовательно, формирователями новых промежуточных артефактов.

3.12 Структура программы

Программа состоит из множества программных артефактов, взаимосвязанных между собой по контексту, наполнение которого определяется стартовой функцией. Эта функция запускается в начале выполнения программы и содержит вызовы других функций, использует различные типы данных и константы. Исходя из этого совокупность всех артефактов, доступных из стартовой функции и определяет общую структуру программы.

Следует отметить, что все артефакты являются независимыми программными объектами. Они описываются и формируются независимо друг от друга и хранятся в виде отдельных сущностей в общей таблице исходных текстов артефактов. При компиляции для каждого из артефактов формируется его промежуточное представление, которое заносится в общую таблицу этих представлений, называемой таблицей реверсивных информационных графов (РИГ). Также в ходе компиляции создается описание артефакта, определяющее его интерфейс, необходимый для взаимодействия с другими артефактами. Это описание добавляется в таблицу экспорта общей базы данных артефактов, образующей пакет (репозиторий) артефактов.

Разработка программного обеспечения заключается в создании одного или нескольких пакетов, образующих приложение и (или) библиотеки функций. Пакеты могут размещаться на различных системах, включая удаленные. Доступ к внешним пакетам осуществляется через директивы импорта.

Примечание. *Пока удовлетворюсь этим описанием, которого явно мало.*

Программа содержит множество описаний, каждое из которых обозначено некоторым именем.

```
обозначенное_описание = {/ идентификатор "<<" /}
        описание {">>" идентификатор}
    | [описание ">>"] идентификатор {/ ">>" идентификатор /}.
описание = функция | константное_выражение | описание_типа.
```

Константное выражение - это любой артефакт языка, вычисляемый на этапе компиляции, и используемый в последующих выражениях как атомарная константа, вектор или группа, атомами которых на самом нижнем уровне вложенности являются константы:

```
константное_выражение = ["const"] значение_константы.
значение_константы = целое | булевское | "!".
```

Пример:

```
pi << const 3.14
```

4 Примеры функций на языке программирования Smile

В разделе представлены примеры функций. Они являются иллюстрацией различных возможностей языка. Помимо этого данные примеры также являются основой для тестирования компонент компилятора и системы программирования. Предполагается что эти примеры будут находиться в наиболее актуальном состоянии, изменяясь в первую очередь по ходу коррекции синтаксиса и семантики языка программирования. Примеры функций предполагается разделить по тематически подразделам, связанным с представлением различной функциональности.

Возможны также дополнительные комментарии, обеспечивающие привязку к различным публикациям.

4.1 Примеры на использование динамически изменяемого асинхронного параллелизма

Первоначально эти примеры были представлены в статье [18, 19]. С течением времени произошло изменение как синтаксиса, так и семантики для некоторых используемых там конструкций. Ниже примеры представлены с использованием актуальной трактовки.

> **Примечание**: На текущий момент модификация примеров связана с их подготовкой в рабочему семинару Хуавей в Сочи (19-20 октября 2023 г.).

4.1.1 Использование потоков

Потоки являются экземплярами соответствующих абстракций. По своей сути экземпляры потоков являются активными объектами, которые могут асинхронно заполняться данными, поступающими из различных функций выполняемой программы. То есть, поток является некоторым общим ресурсом, который не подходит под концепцию функционального программирования. Однако, с другой стороны, нельзя говорить о каждом отдельном потоке как об ограниченном ресурсе, так как поступление в него данных не связано с ресурсными ограничениями. Каждый вновь поступающий элемент «находит» место для своего размещения в очереди элементов, формируемой в порядке поступления во времени.

Такая концепция потока, представляющего единый неограниченный ресурс, определяет свою специфику взаимодействия с ним различных функций программы. Они

не могут напрямую модифицировать поток, за исключением только операций, связанных с добавлением в поток новых данных. Поэтому множественные взаимодействия осуществляются через ссылки на поток. Каждая такая ссылка обеспечивает определенные перемещения, осуществляя тем самым доступ к элементам потока. По сути операции с потоком во многом напоминают упрощенные операции с файлами, доступ к которым осуществляется через дескрипторы файлов или указатели на файлы.

> **Примечание**: Поэтому необходимо ввести описание потоков и описание ссылок на потоки. Использование ссылок, на мой взгляд, формирует более простую концепцию взаимодействия по сравнению с использованием порождения дочерних потоков и наследования в них вновь поступающих данных. Так как внутри потоков запрещена модификация данных (только добавление), то это позволяет достаточно просто организовать доступ, выделив добавление данных в поток как отдельную и общую для него операцию.

Доступ к данным, расположенным в потоке, может осуществляться через ссылки с использованием как одиночной, так и массовой операции интерпретации. В первом случае применяются функции над потоком как активном объектом. Во втором случае используются эквивалентные преобразования массовой операции интерпретации в набор одиночных операций с использованием скрытой от программиста семантикой этих преобразований.

Операции, выполняемые над потоком

> **Примечание**: Скорее всего операции над потоком будут перенесены в описание семантики потоков. Куда — нужно будет разобраться...

Непосредственно на потоком, как объектом данных, выполняется только функция добавления в него нового элемента. При этом поток может быть определен через предварительное описание или непосредственно в программе через именование. Например, описание потока целых через предварительное задание как в функции, так и в глобальном пространстве может выглядеть следующим образом:

```
S1@int<[]
```

Для добавления данных в такие потоки используется специальна операция `push`. Она осуществляет занесение элемента в поток с заданным именем. Эта же операция может использоваться для занесения в поток данных через связанную с ним ссылку. По сути ссылка на поток эквивалентна потоку. Однако над ней можно выполнять операции, связанные с перемещением по элементам потока и чтению данных. Операция `push` может применяться над любым объявленном потоком, имеющим имя и не имеющим дополнительных точек входа. Поток должен находиться в области видимости выполняемой операции. Потоки подобного типа можно считать потоками с внешним доступом. Сигнатура функции, реализующей эту операцию имеет следующий вид:

```
push << func (any, any<[]) -> bool
```

Порождаемой функцией значение информирует о корректности выполнения операции, которая, в силу своей асинхронности, может осуществляться за неопределенное время. Это подтверждение имеет значение `true` при успешной записи элемента в поток или `false`, если запись не прошла или прошла с ошибками. Данный результат можно использовать для анализа выполнения функции или игнорировать.

> **Примечание**: Связь ссылки с потоком — специальная операция над ссылками, которая должна быть описана ниже.

Другой способ задания потоков связан с генераторами их данных, размещенных непосредственно внутри их описания. Размещение генераторов поступающих данных внутри потока локализует источники их доступа. Количество таких генераторов задается при описании потока. Примером такого потока с несколькими точками порождения данных, можно считать следующую конструкцию.

```
S2@<[a, b, c]
```

где `a, b, c` должны порождать элементы одинакового типа. Эти потоки, в зависимости от типа элементов фиксируют их число (в данном случае три элемента). Однако порядок их поступления не фиксируется. Добавление новых значений в подобные потоки также не допускается. Также не допускается доступ из вне с использованием операции `push`. Однако, если внутри подобных потоков определены генераторы данных, то число элементов в них также может меняться.

> **Примечание**: Возникает соблазн добавить `push` к потокам с внутренними генераторами. Но пока воздержусь. Окончательное решение возможно будет принято в ходе реализации потоков и возможностей каждого из механизмов. Может быть будет даже проще отказаться от одного из вариантов из-за дублирования. Какого — пока не знаю...

Ссылки используются для взаимодействия с потоками. Это взаимодействие выливается в чтение элементов в порядке их поступления, переключение на следующий поступивший элемент после обработки текущего. В связи с асинхронностью поступления элементов операции отката или произвольного доступа не реализованы. Но они могут быть реализованы с использованием роя. Связь ссылки с потоком осуществляется с по принципу единственного присваивания, а также с применением при этом разыменования. То есть, можно специально не указывать операцию связывания ссылки с конкретным потоком, осуществив только необходимую подстановку вместо ссылки потока при передаче параметров или указав связываемый поток через именование ссылки. Например:

```
X@<[]
refX@<[*] << X
refY@<[*] << S2
```

Через ссылки с потоками возможно выполнение следующих функций:

- функция `get`, осуществляет чтение элемента из потока;
- функция `is`, предназначенная для проверки на завершение поступивших элементов и отсутствии (empty) поступления новых (конец данных в потоке);

- функция pop, сдвигает ссылку на один элемент потока, имитируя при этом выталкивания головного элемента из потока.

Чтение элемента из потока производится функцией get, которая имеет следующую сигнатуру:

```
get << func any<[*] -> any
```

где any - ключевое слово, обозначающее любой тип. В каждом конкретном случае тип элементов потока определяется его описанием. Функция возвращает значение элемента, первым поступившего в поток. Состояние самого потока при этом не изменяется. Если такой элемент еще не сформировался, функция get ожидает его поступления. При наличии в очереди потока нескольких элементов выбирается только один, поступивший в поток первым. При обращении к опустевшему потоку порождается ошибка EMPTY_ERROR.

Возможность некорректного обращения к опустевшему потоку требует, перед доступом к нему, осуществить предварительную проверку на наличие в нем элементов (по аналогии с проверкой признака конца файла). Это также обуславливается тем, что количество элементов, которые порождает поток, может быть заранее неизвестно. Проверка потока на то, что данные в нем еще порождаются осуществляется функцией is, имеющей следующую сигнатуру:

```
is << func any<[*] -> bool
```

Функция возвращает булево значение true, если поток еще может формировать данные или уже содержит их. В противном случае возвращается false.

Перед тем как прочитать из потока следующий элемент необходимо убрать уже прочитанный. Для этого используется функция pop. При попытке выполнить эту функцию для ссылки достигшей конца потока формируется ошибка EMPTY_ERROR. Функция имеет следующую сигнатуру:

```
pop << func any<[*] -> any<[]
```

То есть, функция возвращает модифицированную ссылку уже без обработанного элемента, что полностью соответствует принципу неизменности (immutable), свойственному языкам функционального программирования.

Суммирование элементов, поступающих в поток

В качестве примера использования функций работы с потоком можно рассмотреть нахождение суммы элементов:

```
sum << func X@float<[*] -> float {
    isEmpty << X:is:not;
    isEmpty^({(X:get,X:pop:sum):+}, 0.0):return
}
```

Проверка `X:is` порождает булевское значение `true/false`, которое используется одиночным оператором интерпретации в качестве селектора. В связи с тем что булево значение `true` соответствует 1, а `false` эквивалентно 1, первым в селекторе вариантов продолжения вычислений должно располагаться ложное значения. Для придания общепринятого порядка в примере используется инверсия результата определения заполненности потока (функция `not`). Поэтому при истинном значение выбирается первый (нулевой) элемент кортежа, запускающий левую рекурсию для функции sum. Значение `false` формируется, когда поток завершен. В этом случае возвращается значение `0.0`. При обратном ходе рекурсивного процесса осуществляется суммирование элементов.

Занесение информации в поток из различных источников

В представленном выше примере суммирования элементов потока по сути реализована левая последовательная рекурсия, так как накопление суммы осуществляется при обратном ходе путем сложения очередного элемента потока с накопленным промежуточным значением. В работе [22] показано, что асинхронный список через последовательные рекурсивные вызовы позволяет реализовать суммирование, параллелизм которого изменяется в зависимости от временных соотношений между интенсивностью поступления данных и скоростью их обработки. Теоретически он может быть эквивалентным каскадной свертке. Использование потоков в языке программирования Smile позволяет написать аналогичную функцию. При этом наличие возможности создавать наполняемые хранилища обеспечивает компактное занесение в поток не только исходных данных, но и промежуточных результатов. Соответствующая функция суммирования значений, поступающих в поток, выглядит следующим образом:

```
// Асинхронное суммирование элементов потока
// с внесением в него результатов промежуточных вычислений
sum << func X@float<[*] -> float {
  // Проверка, что данные в поток еще могут поступить
  isEmptyFirst << X:is:not;
  isEmptyFirst^(
    {block{ // Первый элемент есть
      a << X:get; // Элемент выбирается из потока
      Y << X:pop; // Убирается первый элемент создается новая ссылка
      isEmptySecond << Y:is:not; // Проверка наличия второго элемента
      notEmptySecond^(
        {block{
          // При наличии его можно сложить с первым,
          // который вытолкнуть из потока.
          // После чего втолкнуть в него результат сложения
          // и снова вызвать сумму
          ((a, Y:get):+, Y):push; // И переслать в тот же поток через Y
```

```
          // Также создать новую ссылку без второго элемента
          // и рекурсивно продолжить вычисления
          Y:pop:sum:break     // выход из блока
        }},
          // В противном случае в потоке только один элемент,
          // значение которого и является суммой
          a
      ):break
    }},
      // При отсутствии данных возвращается 0
      0.0
  ):return
}
```

В этой ситуации функция, при наличии хотя бы двух элементов в потоке, суммирует их. Полученная сумма через ссылку пересылается в этот же поток. Процесс рекурсивно повторяется для вновь поступающих элементов, чередующихся с промежуточными вычислениями сумм до момента, когда в потоке останется только одна величина, которая и является окончательной суммой.

Недетерминированное поведение потока при выполнении асинхронных вычислений

Использование потоков, позволяет организовывать асинхронные вычисления с динамически изменяемым параллелизмом, зависящим от временных соотношений между интервалами поступления данных в поток и скоростью их обработки функциями, взаимодействующими с потоком. Однако высокая вероятность того, что порядок поступления аргументов не будет совпадать с порядком получения результатов на выходе, не позволяет во многих случаях организовать детерминированные и предсказуемые вычисления. В качестве такого примера можно рассмотреть вычисление массива данных, поступающих из входного потока, а после обработки направляющихся в выходной поток. Пусть для вычислений используется формула:

```
y[i] = sin(x[i])*sin(x[i])+cos(x[i])*cos(x[i])
```

При использовании потоков в качестве промежуточных хранилищ результатов без особых сложностей организуются конвейерные вычисления (при соответствующих временных соотношениях). Однако, в связи с возможным отличием времени выполнения функций над элементами потоков, корректные последовательности значений в результирующем потоке могут быть не получены. Эта ситуация может быть описана следующим кодом:

```
SumSin2Cos2Stream << func X@float<[*] -> float<[] {
  result@float<[];
  (X, result):GetStreamResult >> ok;
```

```
    result:ok:return
  }
```

где:

```
  GetStreamResult << func (arg@float<[*], result@float<[*])->signal {
    // Проверка потока на возможное поступление данных
    isEmpty << arg:is:not;
    isEmpty^(
      // Занесение результата в выходной поток
      // после добавления в него данных
      {block {
        x << arg:get;    // Получение элемента из потока
        s << x:sin; Sin2 << (s,s):*; // Синус в квадрате
        c << x:cos; Cos2 << (c,c):*; // Косинус в квадрате
        // Вычисление текущего значения с передачей в выходной поток
        ((Sin2,Cos2):+, result):push;
        // Убирается обработанный элемент из потока
        // и переход к обработке следующего элемента
        (arg:pop, result):GetStreamResult}:break
      },
      // Сигнал без заполнения результата,
      // если данные в поток больше не поступают
      !
    ):return
  }
```

Функция `SumSin2Cos2Stream`, получает данные из входного потока через ссылку `X`. Результат вычислений через ссылку на поток возвращается из функции. Сам поток реализован внутри функции через хранилище `result`, а накопление в нем результатов вычислений происходит в функции `GetStreamResult`, в которую он передается в качестве параметра. Передача потока в качестве результата по значению предполагает его копирование во внешнюю среду аналогично тому, как осуществляется передача по значению в современных языках программирования. Возможна также передача по ссылке, если требуется сделать выходной поток внешним хранилищем.

Функция `GetStreamResult` производит основные вычисления для первого текущего элемента, поступившего во входной поток. Полученное значение суммы передается в поток с использованием функции `push`). Одновременно с этим происходит рекурсивный вызов функции `GetStreamResult`, в которую ссылка на входной поток передается уже без учета первого аргумента. Блок `block` используется для локализации группы операторов, из которой только один возвращает результат посредством выполнения функции `break`. Данные в блок поступают через имена, описанные вне его.

При передачи результатов вычислений в новый поток порядок поступления элементов может изменяться относительно первоначального потока, что ведет к появлению недетерминированности вычислений и некорректному результату за счет изменения порядка элементов. Пример показывает, что необходимо расширить модель вычислений конструкциями, обеспечивающими сохранение порядка следования данных и при этом поддерживающие асинхронные взаимодействия.

Применение массовой операции интерпретации с потоками

При использовании массовой операции интерпретации осуществляется обработка всех элементов потока до тех пор, пока не сформируется признак завершения потока. Специальна проверка на завершение потока в данном случае не нужна. Результатом вычислений массовой операции интерпретации, в соответствии с ее семантикой, является выходной поток.

Например, вычисление функции `sin` над всеми элементами входного потока `X` с формированием на выходе потока `Y` на языке Smile можно записать следующим образом:

```
X::sin >> Y ,
```

где поток `X` предварительно описан следующим образом: `X@float<[]`. Символ `@` отделяет имя используемой сущности от ее типа. Автоматически формируемый на выходе поток имеет такой же тип, что и тип результата функции `sin` которая определяется сигнатурой:

```
sin << func float -> float
```

При этом порядок формирования результатов в выходном потоке, обозначенном через `Y` может отличаться от порядка поступления аргументов в поток `X`.

Потоки могут передаваться в функции в качестве параметров. Функция вычисления синуса для всех элементов потока в Smile будет выглядеть следующим образом:

```
sinStream << func X@float<[] -> float<[] {
    X::sin:return
}
```

Примечание: *Кстати. Имеет смысл подумать о реализации простого текстового терминала на основе роя...*

4.2 Использование роя для организации асинхронных вычислений

4.2.1 Организация упорядоченных данных с сохранением порядка следования

Для сохранения порядка следования при обработке данных необходимо использовать контейнерные типы, обеспечивающие асинхронное формирование отдельных

элементов. В ФПМПВ такой сущностью является параллельный список. Однако он поддерживает выполнение только массовых операций над его элементами и не допускает обработки самого списка как единого аргумента. В СТМФППВ вводятся расширения, обеспечивающие поддержку необходимой функциональности. Вместо параллельного списка используется рой, который, наряду с массовыми операциями, как и поток, допускает свое использование в качестве единственного аргумента

Спецификой предлагаемой СТМФППВ и разрабатываемой на ее основе статически типизированного языка ФПП программирования Smile является наличие информации о типах во время компиляции. Это ведет к изменению алгебры эквивалентных преобразований и семантики многих базовых операций, ориентированных не на интерпретацию исходной программы, а на генерацию кода для целевых архитектур. В частности запрещается непосредственная вложенность роев, что облегчает анализ аргументов оператора интерпретации во время компиляции и позволяет определить, является ли функция массовой над всеми элементами роя или это функция над всем роем. Ряд преобразований роя для использования в массовых операциях может происходить уже во время компиляции. Для представления роя используется список из элементов, заключенных в квадратные скобки.

Передача роев, являющихся активными объектам, в функции и возврат их из функций в основном осуществляется, как и для потоков, через ссылки. Синтаксис ссылки на рой в Smile имеет следующий вид:

```
ссылка_на_рой = имя_типа_элементов "[*]".
```

Используя эти ссылки можно написать следующий вариант функции одновременного упорядоченного вычисления синуса для всех элементов роя с использованием массового оператора интерпретации:

```
sinSwarm << func X@float[]->float[] {
    X::sin:return
}
```

В этой ситуации непосредственное использование роевых функций позволяет избавиться от дополнительных преобразований и синхронизации данных как внутри создаваемых функций, так и при их использовании:

```
[0.10, 2.1, 0.33, 1.43]:sinSwarm => [0.0998, 0.8632, 0.324, 0.9901]
```

Компилятор, анализируя тип аргумента функции `sinSwarm`, без проблем может распознать, что она принимает весь рой, а не применяется к каждому из его элементов.

4.2.2 Использование роя для упорядоченной асинхронной обработки потока

В отличие от потоков каждый даже частично сформированный рой имеет предопределенный размер. Поэтому его можно вычислить в любой момент, используя функцию `size`, сигнатура которой описывается следующим образом:

```
size << func any[*] -> int
```

Например:

```
[10,21,33,43]:size => 4
```

Нумерация элементов роя, как и вектора, начинается с единицы. Сами элементы роя формируются асинхронно. При этом поступление каждого из них сопровождается выдачей в связанный с ним оператор интерпретации сигнала, информирующего о формировании очередного значения по определенному индексу. Эти индексы можно упорядочить в порядке поступления и, следовательно, осуществить последовательную выборку отдельных элементов по ним. То есть, можно организовать итератор, делающий обход элементов роя по мере их появления. В отличие от обхода элементов потока, в которых обращение идет непосредственно за созданными элементами, в рое ключевую роль играет получение значения индекса. Для его получения предлагается использовать функцию `get`, которая имеет для роя следующую сигнатуру:

```
get << func any[*] -> int
```

То есть, возвращается индекс элемента, поступившего в рой первым.

Для перехода к следующему индексу, используется функция `pop`. Она возвращает новую ссылку на тот же рой но уже без предшествующего индекса:

```
pop << func any[*] -> any[*]
```

Таким образом можно перебрать все элементы роя в порядке их формирования. В случае, когда через ссылку произойдет перебор всех элементов роя (в порядке их поступления), функция `get` возвращает **отрицательное значение индекса**, которое по сути и определяет завершение обхода.

Помимо этого рой, как и поток, может использоваться для последовательной обработки асинхронно поступающих данных с сохранением при этом порядка следования элементов на выходе. Это позволяет переписать функцию нахождения сумм квадратов синусов и косинусов роя таким образом, что она обеспечивает правильную последовательность результатов на выходе:

```
SumSin2Cos2Swarm << func X@float[*]->float[] {
    l << X::size;
    result@float[l];
    (X, result):GetSwarmResult >> ok;
    result:ok:return
}
```

Для накопления данных функция использует дополнительное роевое хранилище `result`, которое заполняется с использованием принципа единственного присваивания. То есть формируется такой код, который позволяет записать данные по одному и тому же индексу не более одного раза. При нарушении этого правила происходит формирование ошибки `SINGLE_ERROR` с прерыванием выполнения программы. Через ссылку хранилище передается в функцию `GetSwarmResult`, обеспечивающую его заполнение, после чего полученного значение возвращается из функции `SumSin2Cos2Swarm`. Само вычисление осуществляется в функции `GetSwarmResult`:

```
GetSwarmResult << func (X@float[*],Y@float[*])->signal {
  i << X:get; // Получение индекса элемента из X
  if << (i,0)<=; // Проверка наличия элементов
  if^(
    {block {
      s << X:i:sin; Sin2 << (s,s):*; // Синус в квадрате
      c << X:i:cos; Cos2 << (c,c):*; // Косинус в квадрате
      // Вычисление текущего значения с передачей в выходной рой
      // по полученному индексу
      ((Sin2,Cos2):+, Y[i]):push;
      // Убирается обработанный индекс из ссылки на рой
      // и переход к обработке следующего элемента
      (X:{i:signal}:pop, Y):GetSwarmResult}:break
    },  // Занесение результата во второй рой
      // Сигнал, формируемый при завершении вычислений без заполнения,
      // если значение индекса = 0
      !
  ):return
}
```

Первоначально в данной функции вычисляется индекс первого поступившего в рой `X` элемента. Если значение не равно 0, то получен очередной индекс, который непосредственно используется для выборки из роя `i`-го элемента, после чего над ним производится вычисление суммы квадратов синуса и косинуса. Полученное значение заносится через ссылку `Y` на `i`-е место. Отличие от занесения элемента потока заключается в том, что для роя дополнительно указывается индекс в квадратных скобках. Вычисления рекурсивно повторяются до полного заполнения хранилища `result`, передаваемого в данную функцию через ссылку `Y`.

> **Примечание**: Наверное стоит для роя также описать отдельно операции занесения и перемещения, как это выше сделано для потока. Конечно все это нужно будет перенести в семантику языка и добавить простые примеры, демонстрирующие различные ситуации.

4.2.3 Прямое обращение к элементам роя

Наряду с обработкой элементов роя в порядке их поступления возможен и непосредственный доступ по индексу. В этом случае, если элемент еще не поступил, происходит его ожидание. Во время ожидания можно инициировать выборку других элементов, используя для этого параллельно выполняемые рекурсивные вызовы. Недостатком такого подхода является возможность появления множества параллельных ветвей, ожидающих поступления данных. Однако при обработке данных, поступающих из нескольких роев, такой подход облегчает синхронизацию вычислений. Сигнатура функции доступа по индексу имеет следующий вид:

```
base_function{целое} << func any[*] -> any
```

В данном случае в качестве функции используется целое число в диапазоне от 1 до размера роя. Если число не попадает в этот диапазон, то возникает ошибка BOUND_ERROR, вызывающая прерывание в работе программы.

В качестве примера рассмотрим скалярное перемножении данных, поступающих в два независимых роя. Функция ScalMultSignal осуществляет вычисления, принимая в качестве аргументов эти два роя через ссылки X и Y. Помимо этого она получает ссылку R на рой, собирающий результаты, а также число, определяющее количество элементов в роях. Последнее используется в качестве индекса для обращения к элементам.

```
//  функция, непосредственно выполняющая скалярное умножение роев
ScalMultSignal <<
    func (X@float[*], Y@float[*], R@float[*], L@int)->signal {
  if << (L,0):<=;
  if^(
    {block{
      ((X:L, Y:L):*, R[L]):push;
      (X, Y, R, L:--):ScalMultSignal:break
    }},
      // Завершение перебора
      !
  ):return
}
```

Перемножение элементов с одинаковыми индексами осуществляется пока передаваемое значение индекса не изменится до нуля операцией «--», формирующей значение на единицу меньше предыдущего. Рекурсивный вызов осуществляется сразу же после раскрытия задержки, охватывающей блок, независимо от того, будет или нет выполнена операция умножения.

Окончательная функция предоставляет интерфейс для взаимодействия с другими функциями:

```
// Функция, используемая для перемножения роев.
// Предполагается, что размер роев одинаков
ScalMult << func (X@float[*], Y@float[*]) -> float[] {
    L << X:size;
    result@float[L]; // Хранилище результатов
    ok << (X, Y, R, L):ScalMultSignal;
    result:ok:return
}
```

4.2.4 Конвейеризация асинхронных потоковых вычислений

Организация взаимодействия функций на основе передачи между ними потоков и роев позволяет организовать вычисления, обеспечивающие одновременное

выполнение функций, взаимосвязанных между собой. В качестве примера можно рассмотреть функцию векторного произведения с использованием функций скалярного произведения двух векторов `ScalMult` и нахождения суммы элементов потока `sum`:

```
VecMult << func (X@float[*], Y@float[*]) -> float {
  (X, Y):ScalMult:stream:sum:return
}
```

Функция принимает два роя, над которыми осуществляется выполнение скалярного произведения. По мере того, как на выходе функции `ScalMult` формируются результаты перемножения отдельных пар элементов, они поступают в поток, связанный со входом функции `sum`. Конвейеризация в данном случае формируется автоматически в зависимости от темпа поступления исходных данных и скорости выполнения операций внутри функции `VecMult`.

4.2.5 Асинхронное вычисление факториала

Представлено конвейерное зацепление при вычислении факториала. Пример для доклада на семинаре Хуавей в Сочи.

Для порождения чисел от 2 до N используется функция `Generator`, порождающая на выходе асинхронно заполняемый поток:

```
Generator << func (first@int, last@int) -> @int<[] {
  data@int<[];
  numbers << (last,first):-; // осталось чисел
  selector << (numbers,0):(<,=,>):?   // (номера истинных значений)
  // Только одно истинное значение из трех вариантов: 0, 1, 2
  (
    true,      // Нет значений - подтверждение успеха
    {
      (first:RndDelay,data):push  // Совпали. Любое в поток.
    },
    {
      // Уменьшение интервала и расчет по новой
      (
        ((first:RndDelay,data), (last:RndDelay,data))::push:and,
        (fist:++, last--, data):Generator
       ):and;
    }

  ) >> ok
  data:return; // Возрат потока, формируемого в ходе вычислений
}
```

Для имитации разной скорости передачи данных в поток используется функция `RndDelay` обеспечивающая некоторую задержку во времени с передачей своего аргумента (она не показана). Можно отметить, что выходной параметр данной функции не нужен. То есть, она завершает работу по заполнению потока. Момент завершения ее самой роли не играет. То есть, она похожа на зомби по своему поведению. Отследить ее завершение может ОС по выполнению функции `return`.

Перемножение элементов потока осуществляет до тех пор, пока не будут обработаны все поступившие в него данные. В результате формируется произведение всех элементов, поступивших поток, включая промежуточные произведения

```
// Асинхронное перемножение элементов потока
// с внесением в него результатов промежуточных вычислений
Product << func data@int<[*] -> int {
  // Проверка, что данные еще могут поступить
  isEmptyFirst << data:is:not;
  isEmptyFirst^(
  {block{ // Первый элемент есть
    i << data:get; // Элемент выбирается из потока
    next_data << data:pop; // Следующий элемент стал первым
    isEmptySecond << next_data:is:not; // Есть ли 2-й элемент
    isEmptySecond^(
      {block{
        // Перемножение первого и второго
        // И в тот же поток после имитации задержки
        ((i, next_data:get):*, data):RndDelay:push;
        // Сдвиг и перемножение до получения результата
        next_data:pop:Product:break
      }},
      // В противном случае в потоке только один элемент,
      // значение которого и является результатом
      i
    ):break
  }},
    // При отсутствии данных возвращается 1
    1
  ):return
}
```

Окончательная функция собирает вместе два независимых процесса, объединенных между собой потоком, обеспечивающим пересылку между ними асинхронно формируемых данных. Предполагается, что поток, формируемый генератором передается в поток описанный в данной функции по значению. В принципе возможна реализация, когда эта передача может осуществляться в асинхронном режиме и и по сути являться наложением одного потока на другой.

```
Factorial << func N@int -> int {
    data@<[int];
    (2, N, data):Generator >> ok
    0 >> fall
    data:Product >> result
    (fall, result):ok:return
}
```

Формируемое в ходе генерации потока значение об успешности записи в поток используется для дополнительной проверки, что все данные, порождаемые генератором дошли успешно. При успешной генерации на выходе функции формируется результат. В противном случае на ее выходе формируется нулевое значение.

Заключение

Продолжение следует...

Литература

[1] Штейнберг, Б. Я. Преобразования программ — фундаментальная основа создания оптимизирующих распараллеливается компиляторов / Б. Я. Штейнберг, О. Б. Штейнберг // Программные системы: теория и приложения. – 2021. – Т. 12. – № 1(48). – С. 21-113. – DOI 10.25209/2079-3316-2021-12-1-21-113. – EDN FZFEPX.

[2] Legalov A.I., Vasilyev V.S., Matkovskii I.V., Ushakova M.S. A Toolkit for the Development of Data-Driven Functional Parallel Programmes. In: Sokolinsky L., Zymbler M. (eds) Parallel Computational Technologies. PCT 2018. Communications in Computer and Information Science, vol 910. Springer, Cham, pp 16-30. DOI https://doi.org/10.1007/978-3-319-99673-8_2.

[3] Pierce Benjamin C. Types and Programming Languages // The MIT Press, 2002.

[4] Алеева В.Н., Алеев Р.Ж. Применение Q-детерминанта численных алгоритмов для параллельных вычислений // Параллельные вычислительные технологии – XIII международная конференция, ПаВТ'2019, г. Ростов-на-Дону, 2–4 апреля 2019 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2019. С. 133-145. [Текст в формате pdf: `http://omega.sp.susu.ru/pavt2019/short/006.pdf`

[5] High-Performance Reconfigurable Computer Systems with Immersion Cooling (p. 62-76).

[6] Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ. – Вычислительные технологии, № 1 (10), 2005. С. 71-89.

[7] Легалов А.И. Об управлении вычислениями в параллельных системах и языках программирования — Научный вестник НГТУ, № 3 (18), 2004. С. 63-72.

[8] Деннис Дж.Б, Фоссин Дж.Б., Линдерман Дж.П. Схемы потока данных. — Теория программирования, Т2, 1972, с. 7-43.

[9] Касьянов В.Н., Стасенко А.П. Язык программирования Sisal 3.2 // Методы и инструменты конструирования программ. — Новосибирск: ИСИ СО РАН, 2007. — С. 56–134.

[10] Каляев И.А., Левин И.И. Реконфигурируемы вычислительные системы на основе ПЛИС. — Ростов-на-Дону: Издательство ЮНЦ РАН, 2022. — 475 с.

[11] Перепелкин В.А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах — журнал "Проблемы информатики 2020, № 1. — с.66-74. DOI: 10.24411/2073-0667-2020-10004

[12] Легалов А. И. Функциональный язык для создания архитектурно-независимых параллельных программ. // Вычислительные технологии № 1 (10). 2005. С. 71–89

[13] Легалов А.И., Редькин А.В., Матковский И.В. Функционально-потоковое параллельное программирование при асинхронно поступающих данных. // Параллельные вычислительные технологии (ПаВТ'2009): Труды международной научной конференции (Нижний Новгород, 30 марта – 3 апреля 2009 г.). — Челябинск: Изд. ЮУрГУ, 2009. — С. 573–578. (Электронное издание)

[14] Romanova D. S., Nepomnyashchiy O. V., Ryzhenko I. N., Legalov A. I., Sirotinina N. Y. Parallelism reduction method in the high-level VLSI synthesis implementation. // Trudy ISP RAN/Proc. ISP RAS. 2022. Vol. 34, No. 1. P.59–72. DOI: 10.15514/ISPRAS-2022-34(1)-5

[15] Васильев В. С., Легалов А. И., Зыков С. В. Трансформация функционально-потоковых параллельных программ в императивные. // Моделирование и анализ информационных систем. № 2 (28). 2021. С. 198–214. DOI: 10.18255/1818-1015-2021-2-198-214

[16] Backus, J. Can programming be liberated from von Neuman style? A functional stile and its algebra of programs. CACM 21(8), 613–641 (1978). DOI: 10.1145/359576.359579

[17] Alexander Legalov, Igor Legalov, Ivan Matkovskii. Specifics of Semantics of a Statically Typed Language of Functional and Dataflow Parallel Programming - Scientific Services & Internet 2019. CEUR Workshop Proceedings, Vol. 2543. P. 274–284. DOI: 10.20948/abrau-2019-08.

[18] Легалов А.И., Матковский И.В., Ушакова М.С., Романова Д.С. Динамически изменяющийся параллелизм с асинхронно-последовательными потоками данных. Моделирование и анализ информационных систем. 2020;27(2):164–179. DOI: 10.18255/1818-1015-2020-2-164-179

[19] Legalov A. I., Matkovskii I. V., Ushakova M. S., Romanova D. S. Dynamically Changing Parallelism with Asynchronous Sequential Data Flows // Automatic Control and Computer Sciences. 2021. Vol. 55, No. 7. P. 636–646. DOI: 10.3103/S0146411621070105

[20] Легалов А.И., Чуйкин Н.К. Поддержка статической типизации в функционально–потоковой модели параллельных вычислений // Параллельные

вычислительные технологии – XVII всероссийская конференция с международным участием, ПаВТ'2023, г. Санкт-Петербург, 28–30 марта 2023 г. DOI: 10.14529/pct2023. С. 173–185. Электронная версия статьи размещена по адресу: http://omega.sp.susu.ru/pavt2023/short/024.pdf

[21] Legalov, A. I., & Chuykin, N. K. (2023). The Semantic Model Features of the Statically Typed Language of Functional-dataflow Parallel Programming. Supercomputing Frontiers and Innovations, 10(2), 32–45. DOI: 10.14529/jsfi230203

[22] Легалов А.И. Использование асинхронно поступающих данных в потоковой модели вычислений. / Третья сибирская школа-семинар по параллельным вычислениям. / Томск. Изд-во Томского ун-та, 2006. С 113–120.