Computer Science Center
Yandex ML & Data Management Lab

Контрактное программирование как способ задания и оптимизации вычислительных графов для распределённых потоковых систем

Руководитель: Трофимов Артём
Студент: Стоян Андрей

СПб 2020

Содержание

1 Введение

Приложения, обрабатывающие большие объёмы данных, используют фреймворки, позволяющие производить вычисления с состоянием над ограниченными и неограниченными потоками данных, такие как, например, Apache Flink [1].

Подобные фреймворки обычно запускаются на кластере машин, получают запросы на обработку данных от приложений, берут данные из источников, производят обработку в соответствии с запросом и формируют результат в виде записи в базе данных или в виде нового потока (см. рис. 1).

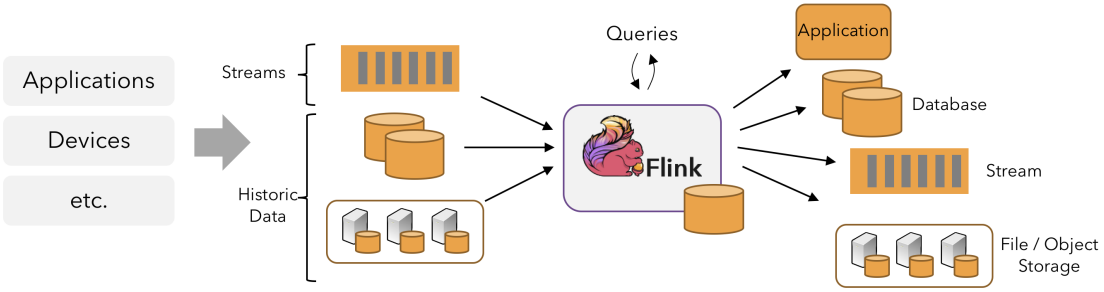

Рис. 1: Использование Apache Flink.

Запросы задаются в виде вычислительного графа (пайплайна), который представляет собой ориентированный граф без циклов. Пример задания такого графа представлен на рисунке 2.

Копии пайплайна распределяются между машинами для параллельной обработки данных. Часть операций не содержат состояния и не требуют дополнительных усилий, чтобы быть запущенными в нескольких экземплярах и работать параллельно. Некоторые операции же, такие как агрегация, требуют наличия изменяемого состояния и специального перераспределения (решардирования) данных между их экземплярами (чтобы записи с одним и тем же ключём подсчитывались только на одной машине, например, и не было необходимости производить агрегацию по машинам). Пример показан на рисунке 3.

В данной работе предлагается новый способ задания вычислительных графов для распределённых потоковых систем, который позволяет упростить программирование комплексных пайплайнов и производить их автоматическую оптимизацию.

Все материалы работы можно найти в GitHub репозитории проекта.

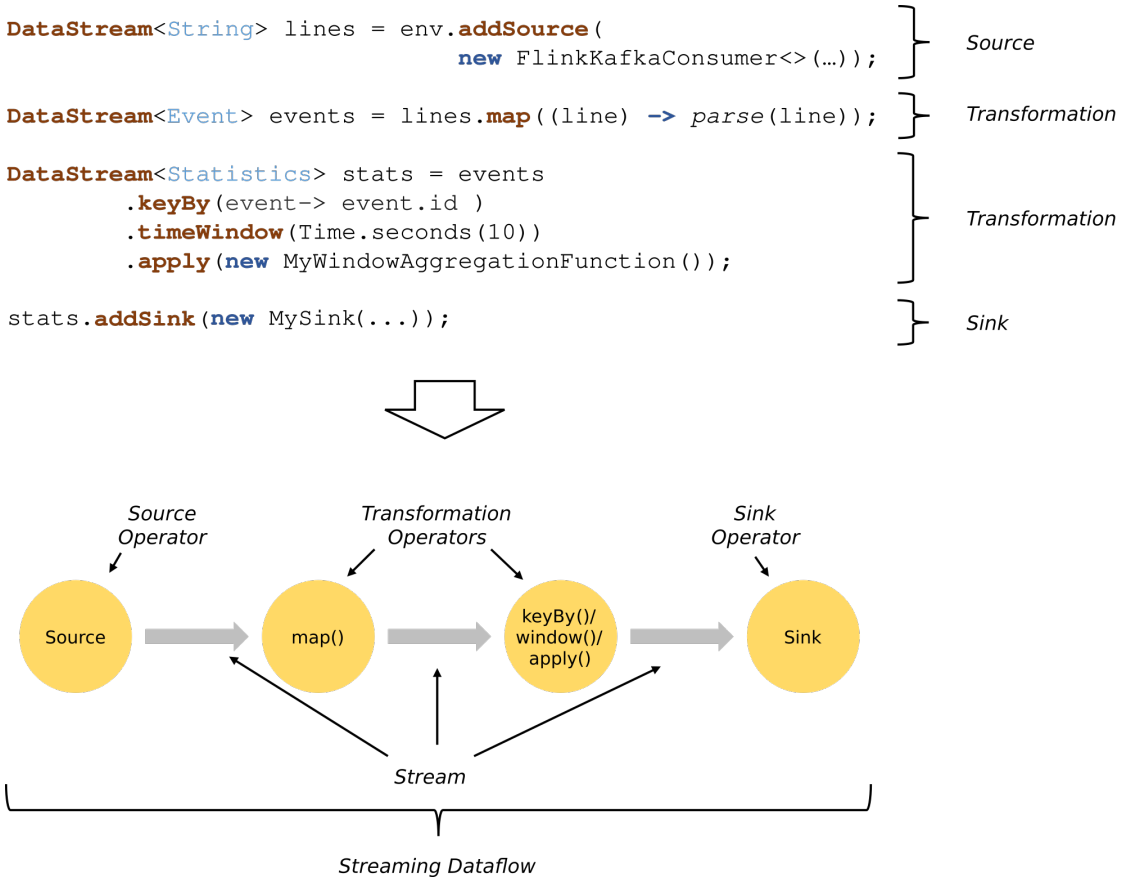

Рис. 2: Пример вычислительного графа для Apache Flink.

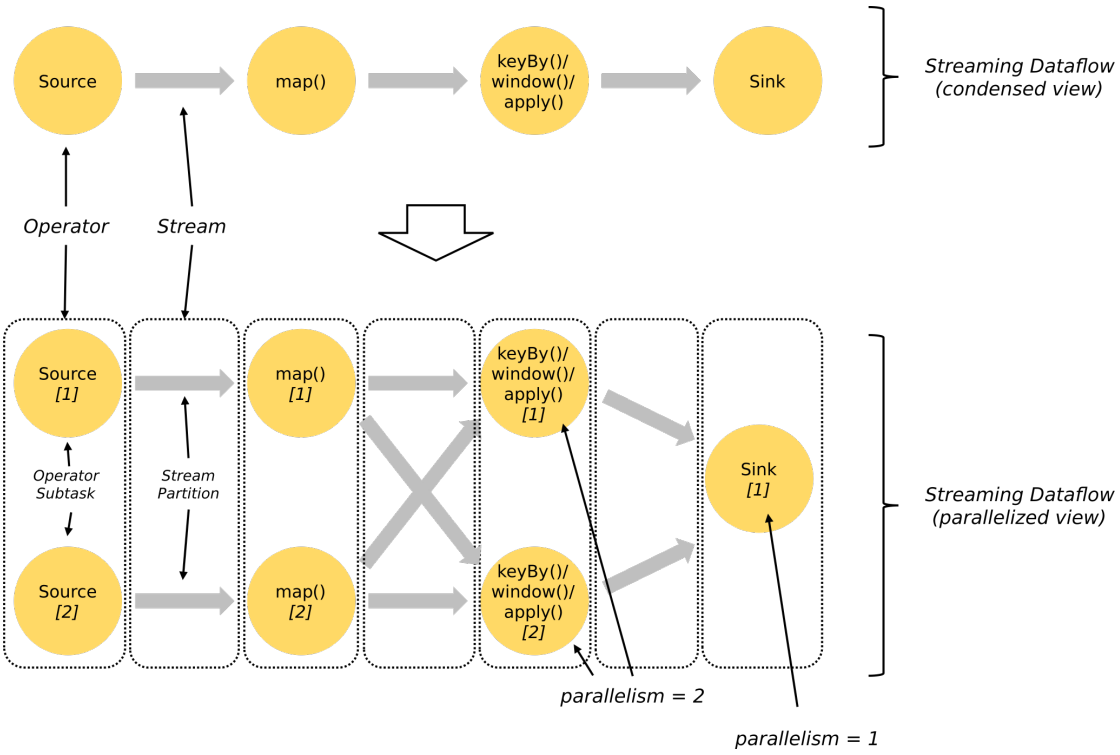

Рис. 3: Распределение данных между экземплярами операций одного и того же пайплайна, размещёнными на разных машиных. Операция *map()* не имеет состояния, и её два экземпляра параллельно обрабатывают один и тот же поток данных, не влияя друг на друга. Операция *apply()* же имеет изменяемое состояние, поэтому необходимо производить перераспределение данных (решардирование) между её экземплярами с помощью *keyBy()*.

2 Проблема

Современные фреймворки предоставляют два основных способа задания графов: композиция примитивных операций (*Map*, *Reduce* и т.д.) и SQL запросы (диалект Streaming SQL, для потоковой обработки [2]). На рисунке 4 представлена иерархия Apache Flink API.

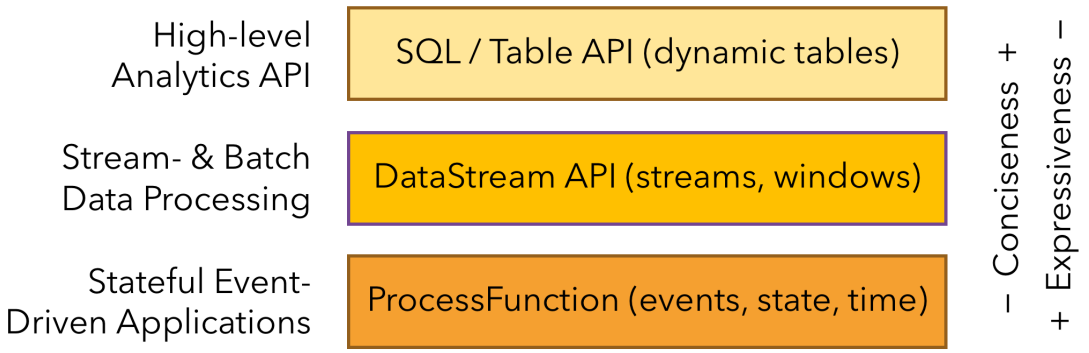

Рис. 4: Иерархия Apache Flink API. Видим, что фреймворк предоставляет три уровня интерфейсов: от самого выразительного, *ProcessFunction*, до самого лаконичного, SQL. *ProcessFunction* и *DataStream* позволяют определять операции и составлять их композиции.

Пусть мы хотим записать граф, схематично изображенный на рисунке 5.

Когда мы составляем граф как композицию операций (как на рисунке 2), мы однозначно определяем их порядок и не предоставляем фреймворку информации о том, какие операции с какими можно переставлять в целях оптимизации, оставляя пайплайн корректным. Таким образом, возможна только ручная оптимизация.

Диалект Streaming SQL позволяет использовать в SQL запросах неограниченные источники данных. Как и в случае с традиционным SQL, он предназначен для выборки данных из источников и подсчёта статистик, а не для записи комплексной бизнес-логики.

Пусть мы уже имеем написанный и работающий на кластере граф 5. Попробуем добавить этому графу ещё одну функциональность и вернуть на кластер. Постараемся максимально использовать промежуточные значения, вычисленные в исходном графе, чтобы не производить лишних повторных вычислений.

Сейчас для таких модификаций требуется вручную исправлять граф и производить оптимизации (минимизировать количество решардирований, размещать фильтрации как можно ближе к источникам данных и т.д.). В случае

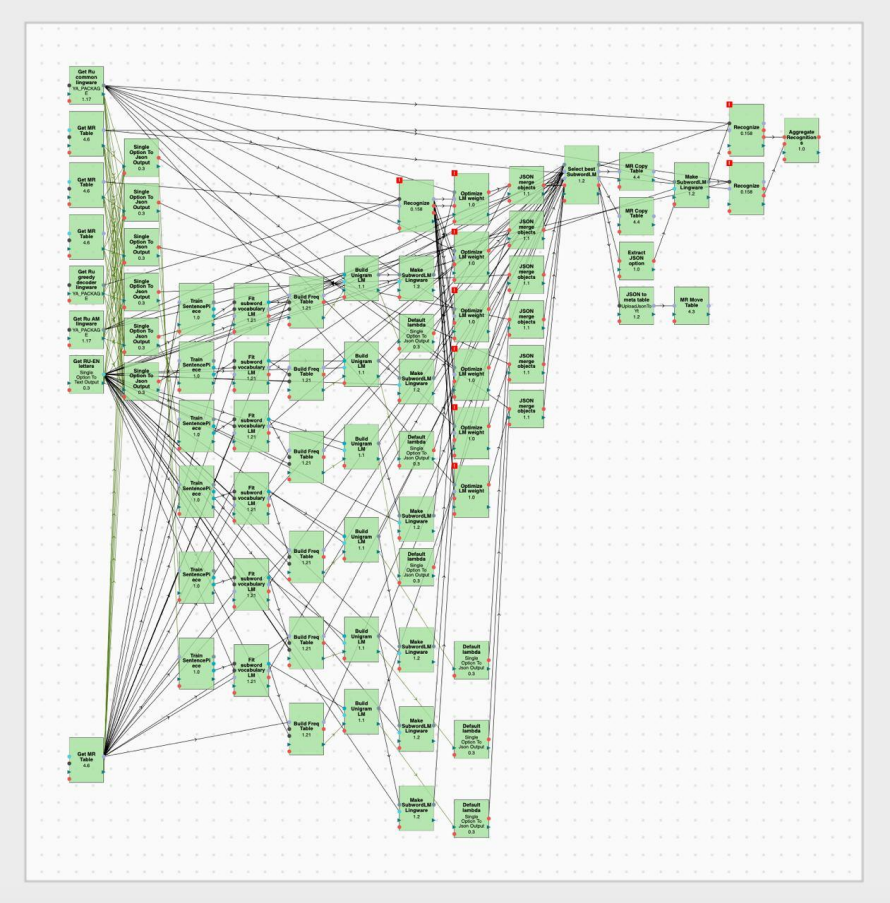

Рис. 5: Графы, производящие обработку данных для реальных приложений могут быть довольно большими.

больших графов это довольно сложная работа, подверженная трудно уловимым ошибкам.

Таким образом, требуется разработать декларативный подход к описанию вычислительных графов, который бы упростил их задание и модификацию, и позволил бы производить автоматические оптимизации путём перестановки операций в графе.

3 Идея

Вершины вычислительных графов могут быть двух типов: **источники** и операции (**трансформации**). Источники не получают данных, только отдают. Операции принимают один или несколько потоков данных и могут порождать поток данных, а так же могут формировать результаты работы графа (в виде записи в базе данных, нового потока для другой системы, и т.д.).

Будем называть **нодами** вершины обоих типов.

Чтобы иметь возможность изменять форму графа, для каждой операции необходимо иметь информацию о требованиях, которые она выдвигает к данным, поступающим на её вход, и гарантии, которые она выставляет для производимых ею данных.

Такие требования и гарантии назовём **контрактами** операции, **входными** и **выходными** соответственно.

Далее будут приводиться листинги на языке Haskell, формализующие вводимые понятия.

```
1   -- InCont - входной контракт.
2   -- OutCont - выходной контракт.
3   -- Пока предположим, что трансформаций арности два будет достаточно.
4
5   -- Вершины вычислительного графа, аннотированные контрактами
6   data Node =
7       Stream OutCont              -- Источник данных.
8     | Tfm1 InCont OutCont         -- Трансформация арности один.
9     | Tfm2 InCont InCont OutCont  -- Трансформация арности два.
```

Зададим множество нод, аннотированных контрактами (**окружение**):

```
1   type Env = Map NodeName Node
```

Определим **семантику** вычислительного графа как подмножество множества нод, элементы которого формируют требуемые от графа результаты работы:

```
1   type Semantics = Set NodeName
```

Наконец, определим **ц-граф**, который перставляет собой пару из окружения и семантики:

```
1   type CGraph = (Env, Semantics)
```

Пусть ц-граф задаёт множество **конкретных графов** (или просто **графов**) следующим образом: граф задаётся данным ц-графом, если он состоит только из нод окружения, содержит каждую ноду семантики один раз, и контракты всех используемых нод удовлетворяются.

Будем использовать ц-графы в качестве нового способа задания вычислительных графов. Каждый конкретный граф содержит ноды, формирующие требуемый результат работы. Все ноды каждого графа получают валидные данные на вход и, следовательно, работают корректно. Таким образом, все операции семантики так же работают корректно, и каждый граф, соответствующий данному ц-графу, выполняет одну и ту же, требуемую, работу.

Процесс от задания ц-графа до запуска задачи на целевом фреймворке выглядит следующим образом:

```
1   -- [Graph] - список графов.
2   -- Точка - оператор композиции функций.
3
4   -- RGraph - конкретный граф в API целевого фреймворка.
5   run :: CGraph -> Runtime RGraph
6   run = graphToRGraph     -- Преобразовать конкретный граф в
7                           -- вычислительный граф целевого фреймворка.
8       . chooseGraph cost  -- Выбрать оптимальный граф в соответствии
9                           -- с функцией оценки cost.
10      . genGraphs         -- Сгенерировать все графы, соответствующие
11                          -- данному CGraph.
12
13  genGraphs :: CGraph -> [Graph]
14  cost :: Graph -> Integer
15  chooseGraph :: (Graph -> Integer) -> [Graph] -> Graph
16  graphToRGraph :: Graph -> Runtime RGraph
```

3.1 Конкретные графы

Будем представлять граф виде множества термов. Результат работы одной ноды может подаваться на вход нескольким нодам, поэтому в операциях будем хранить не сами подтермы, а их идентификаторы.

```
1   data Term =
2       Const NodeName
3     | App1 NodeName TermId
4     | App2 NodeName TermId TermId
5
6   type Graph = Map TermId Term
```

3.2 Контракты

Будем считать, что данные имеют формат записей с именованными атрибутами. Тогда существует два сорта информации о потоке данных: какие ат-

рибуты имеют записи и какие существуют свойства атрибутов и связей между ними.

Введём состояние потока как совокупность информации о нём:

```
1   -- Атрибуты
2   type Attr = AttrName
3
4   -- Свойства
5   type Prop = PropName
6
7   data State = State
8     { attrs :: Set Attr -- Множество атрибутов в потоке.
9     , props :: Set Prop -- Множество свойств в потоке.
10    }
```

Введём входной контракт и правило проверки:

```
1   data InCont = InCont
2     { attrsI  :: Set Attr -- Множество требуемых атрибутов.
3     , propsI  :: Set Prop -- Множество требуемых свойств.
4     , propsI' :: Set Prop -- Множество запрещённых свойств.
5     }
6
7   match :: State -> InCont -> Bool
8   match s c = attrsI c `Set.isSubsetOf` attrs s
9           && propsI c `Set.isSubsetOf` props s
10          && propsI' c `Set.disjoint`  props s
```

Пусть выходной контракт определяет разницу между состояниями входного и выходного потоков. Введём выходной контракт и правило обновления состояния:

```
1   data OutAttrs =
2       AddAttrs (Set Attr) -- К множеству входящих атрибутов
3                           -- добавятся новые из множества.
4     | NewAttrs (Set Attr) -- Множество входящих атрибутов
5                           -- заменится на новое.
6
7   data OutCont = OutCont
8     { attrsO  :: OutAttrs -- Изменения в составе атрибутов
9                           -- относительно требуемых.
10    , propsO  :: Set Prop -- Множество добавляемых свойств.
11    , propsO' :: Set Prop -- Множество удаляемых свойств.
12    }
13
14  update :: State -> OutCont -> State
```

```
15    update s c =
16      State { attrs = attrs'
17            , props = props0 c `Set.union` (props s \\ props0' c) }
18      where
19        attrs' = case attrs0 c of
20          AddAttrs attrs' -> attrs' `Set.union` attrs s
21          NewAttrs attrs' -> attrs'
```

Если если в операцию входят два потока, то чтобы посчитать состояние выхода, состояния входных потоков объединяются:

```
1    union :: State -> State -> State
2    union s1 s2 = State { attrs = attrs s1 `Set.union` attrs s2
3                        , props = props s1 `Set.union` props s2 }
```

Состояние ребра, выходящего из источника совпадают с выходным контрактом этого источника. Если состояние входящих рёбер удовлетворяет входным контрактам операции, то состояние выходящих вычисляется обновлением состояния входа в соответствии с выходным контрактом.

Таким образом, мы умеем для конкретного графа вычислять состояния всех его рёбер и проверять контракты всех его нод.

3.3 Верификация

Заметим, что предложенная система контрактов является также и средством верификации вычислительных графов, подобным логике Хоара, предназначенной для доказательства корректности императивных компьютерных программ. В логике Хоара задаются так называемые тройки Хоара — предусловие, команда и постусловие. Программа считается корректной, если удовлетворяются предусловия и постусловия [7].

4 Вид реализации

Реализация состоит из двух частей: основная, которую предполагается использовать на практике, и прототип, представляющий собой высокоуровневое описание модели контрактов и работы с ней.

4.1 Основная реализация

Основная реализация представлена в виде библиотеки на языке Python, позволяющей использовать ц-графы для задания вычислительных графов в Apache Beam Python API [4].

Проект Apache Beam предоставляет универсальную модель задания вычислительных графов и позволяет запускать графы, написанные в этой модели, в различных рантаймах [3]. Например, на Apache Flink, который мы в качестве примера рассматривали выше.

Python был выбран как язык с динамической типизацией, потому что статическая мешала бы произвольной перестановке пользовательских операций.

Реализацию библиотеки можно найти в GitHub репозитории.

4.2 Прототип

Для реализации прототипа был выбран Haskell — чистый функциональный язык программирования с сильной статической системой типов.

В терминах типов удобно вводить нужные нам понятия, что и было сделано выше. Статическая типизация предоставляет довольно высокие гарантии корректности. А выразительность функционального языка позволяет естественно и довольно формально описывать работу с контрактами.

В прототипе не предполагалось реализовывать тела операций, аннотированных контрактами, поэтому статическая типизация не является проблемой в данном случае.

Реализацию прототипа можно найти в GitHub репозитории проекта.

5 Предварительные результаты

Реализовано в прототипе:

- Контракты в виде структур данных языка Haskell.
- Процедура проверки того, что данный конкретный граф задаётся данным ц-графом.
- Пример задания вычислительного графа контрактами (ц-графом).

Также был создан набросок вида API библиотеки (основной реализации).

6 Дальнейшая работа

Несмотря на объём проделанной работы, реализуемость идеи ещё не доказана:

- Не очевидно существование достаточно быстрого алгоритма перебора всех конкретных графов, задаваемых данным ц-графом. Ожидается, что много вариантов будут отсеиваться как не удовлетворяющие контрактам, и большая асимптотика переборного алгоритма не будет сказываться на практике.
- В основной реализации требуется предоставить универсальный доступ к именованным атрибутам записей для кода пользовательских операций, аннотированных контрактами. Но ключевые операции Apache Beam нередко принимают и возвращают довольно комплексные структуры данных, для которых сложно обеспечить требуемое API [5].

До применения ц-графов на практике требуется ещё решить следующие задачи:

- Рассмотреть большое количество примеров из практики, поправить контракты и способ их записи, чтобы удобно было решать практические задачи.
- Перенести реализацию с прототипа на Python.
- Разработать функцию оценки, чтобы в соответствии с ней выбирать самый оптимальный конкретный граф.

7 Вывод

Таким образом, была развита идея записи вычислительных графов через набор операций, аннотированных контрактами, и сделаны первые шаги в её реализации.

Приложение А: развитие идеи

Первоначальной идеей была следующая: задавать множество конкретных графов, решающих данную задачу как пару из типа требуемого результата вычислений и контекста. **Контекст** — это набор типизированных функций (операций) и констант (потоков).

Тогда каждый конкретный граф задавался бы термом, имеющим тип результата в данном контексте. Имея алгоритм, перебирающий такие термы, мы бы получали множество конкретных графов, решающих поставленную задачу. При наличии функции оценки эффективности конкретного графа, можно было бы выбрать из множества самый эффективный.

При разработке этой идеи возникало множество проблем.

Задача населения типа (inhabitation problem) — существует ли терм, имеющий заданный тип в данном контексте.

Оказывается, что задача населения разрешима только для довольно простых систем типов (например, $\lambda_{\rightarrow}$ или $\lambda\cap$). Если задача населения неразрешима, то заведомо не существует завершающегося алгоритма перебора. Таким образом, наиболее подходящие системы типов, с row polimorphism, нам не подходят [6].

Для оптимизации хотелось бы генерировать как можно больше различных графов, решающих одну и ту же задачу. Но оказывается, что простые системы типов просто не обеспечивают нужной гибкости. Например, есть только один способ расположить операции с типами $A \to B$ и $B \to C$. Чтобы их можно было переставлять местами, они должны быть эндоморфизмами: $A \to A$ и $A \to A$, но тогда у нас нет инструментов, чтобы гибко ограничивать перестановочность. $\lambda\cap$ предоставляет некоторые дополнительные возможности, но их тоже не достаточно.

Таким образом, только с помощью типов не получается добиться нужной гибкости с сохранением возможности ограничивать перестановочность. Закономерно возникает идея расширить типы с помощью refinement types, вроде того, как сделано в Liquid Haskell. Но при попытке добиться больших возможностей в ограничении перестановочности с помощью аннотирования типов предикатами, оказывается, что предикаты начинают нести всю выразительную нагрузку, и в типах отпадает смысл.

Идея с refinement types помогла оторваться от типов и подумать более широко. Типы, аннотации к типам — это всё способы выразить контракт функции, перенести его проверку на этап компиляции.

Таким образом, на самом деле задача стоит в построении конкретных графов по набору операций, аннотированных контрактами. Идея с типами оказалась одним из потенциальных подходов к решению настоящей задачи.

Понятие контракта в свою очередь позволяет решать задачу в терминах

предметной области, начиная с размышлений о том, что такое контракт операции в вычислительном графе. То есть изначальный ход рассуждений вёл от частного к общему, от способа выражать контракты к самим контрактам, которые на самом деле всецело обуславливаются предметной областью. Это не могло привести к положительному результату.

Список литературы

[1] Apache Flink: `https://flink.apache.org/`.

[2] Реализация диалекта Streaming SQL в Apache Calcite: `https://calcite.apache.org/docs/stream.html`.

[3] Apache Beam: `https://beam.apache.org/`.

[4] Apache Beam Python API: `https://beam.apache.org/documentation/sdks/python/`.

[5] Core Beam transforms: `https://beam.apache.org/documentation/programming-guide/#core-beam-transforms`.

[6] Gaster, Benedict R., and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.

[7] Hoare logic: `https://en.wikipedia.org/wiki/Hoare_logic`.