Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

НАПИСАНИЕ УТИЛИТЫ ПОИСКА ДЕКЛАРАЦИЙ И КОНСТРУИРОВАНИЯ ВЫРАЖЕНИЙ ПО ФРАГМЕНТАРНОЙ ТИПОВОЙ СИГНАТУРЕ

Автор: Градобоев Денис Сергеевич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Беляев М.А., без степени _____

Санкт-Петербург, 2024 г.

Обучающийся Градобоев Денис Сергеевич
Группа М34341 Факультет ИТиП

Направленность (профиль), специализация
Информатика и программирование

Консультанты:

а) Машков С.Н., без степени, без звания _____

б) Стоян А.С., без степени, без звания _____

ВКР принята «____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «15» июня 2024 г.

Секретарь ГЭК Штумпф С.А. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Инструменты разработчика позволяют экономить время и решать задачи поддающиеся автоматизации. Одной из часто возникающих задач является поиск функций, выполняющих требуемую операцию, который можно решать разными способами. В работе рассматривается подход, который предполагает, что знание типов, участвующих в преобразовании, может служить информацией достаточной для поиска деклараций, например имён функций или конструированию выражений, отражающей интуицию разработчика.

В цели работы входит рассмотрение условий, необходимых для реализации такого инструмента для произвольного языка программирования, а так же логическое продолжение идеи поиска функции по сигнатуре до конструирования выражений, состоящих из деклараций на основе частично описанного типового пути.

В задачи работы входит:

- разработка синтаксиса языка запросов и его парсинг
- разработка поискового движка
- описание требований для языка, позволяющих использовать этот инструмент
- тестирование полученного решения
- анализ с инженерной и теоритической точек зрения

Новизна работы заключается, во-первых в рассмотрении вопроса без привязки к конкретному языку, то есть по возможности в общем виде, а во-вторых в попытке конструировать выражения, в то время как в паре известных решений реализован только функционал поиска без возможности комбинирования базовых блоков.

Работа разделена на несколько частей. В первой главе сначала даётся общих обзор предметной области с упоминанием фактов и упрощений, необходимых для дальнейшего понимания, и последующее более строгое описание необходимых в дальнешем терминов. Следующая глава содержит непосредственное рассмотрение вопроса, формализацию синтаксиса запросов, описание способа хранения данных и алгоритма поиска. Заключительная глава состоит из подведения итогов, анализа полученного решения и рассмотрения возникших органичений.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Тема работы лежит в домене языков программирования. Данная область обширна, потому вступление содержит лишь краткий обзор, необходимый для дальнейшего понимания.

Программисты в ходе своей деятельности используют множество абстракций, программа — одна из них. В первую очередь под этим понимается исходный код, и то, что он должен делать. Исходный код, в своём наиболее привычном виде, является текстом. Этот текст обладает структурой, которая определяется синтаксисом языка. Синтаксис языка, как совокупность правил, устанавливает возможность комбинации языковых конструктов. Семантика языка, в свою очередь, предписывает, что данный структурированный текст означает, что должен делать или какой результат получить.

Для удобства понимания стоит обозначить популярный современный подход к устройству компиляторов — программ преобразования исходного кода в машино-удобное представление. Можно выделить три логические части: фронтенд, миддленд и бэкенд. Фронтэнд проводит процедуру парсинга, то есть выделяет из исходного кода структуру согласно синтаксису языка. В ходе этого получается абстрактное синтаксическое дерево — граф, являющийся деревом, вершины которого являются языковыми конструктами и содержат информацию, по которой ещё можно восстановить исходный код. Далее производится семантический анализ, в ходе которого выводятся типы, проводится разного рода диагностика о корректности согласованности логики программы. Этапы миддленда преобразуют эту информацию в промежуточное представление, которое «приземляет» языковые абстракции в более специфичный для машины вид. Здесь же проводится часть оптимизаций, хотя они присутствуют практически на всех этапах, однако в данном месте содержится достаточно семантической информации, которая позволяет делать выводы об особенностях программы. За дальнейшую материализацию программы отвечает бэкенд. Он преобразует промежуточное представление в машинный код для целевой платформы, проводит оптимизации, специфичные для исполнения на конкретной архитектуре.

Стоит отметить что это лишь один из возможных способов организации компиляции, тем не менее он вполне соотносится с практикой написания популярных современных решений.

1.1. Типы и декларации

Программист при взаимодействии с исходным кодом часто работает в терминах типов и деклараций. Декларация — это целостный кусок текста, который вводит в контекст новую сущность, которая может быть либо непосредственно использована в месте опреления, или же в других местах по её идентификатору, то есть имени. К декларациям относятся функции, переменные, объявления классов, структур и прочее.

Тип — всмопогательная метка в языке, она определяет способы взаимодействия с объектом. При первом приближении можно выделить следующие разновидности типов:

а) примитивные
   - логический тип (Boolean)
   - целый числа (Integer)
   - числа с плавающей точкой (Float)
   - указатели (Pointer)

б) функциональные

в) номинативные

Примитивные типы, зачастую, являются частью языка, тогда как функциональный и номинативный могут определяться разработчиком. Под номинативными типами следует понимать классы, структуры, перечисления. Функциональные типы обозначают типы функций, то есть множества входных и выходных типов, наиболее популярна запись со стрелкой A -> B, где A - входной тип, а B — выходной.

Листинг 1 – Пример объявления и инициализации переменной типа $Float$

```
Float PI = 3.14
```

Листинг 2 – Пример функции типа Integer -> Boolean

```
function isZero(x: Integer) {
    return x == 0
}
```

я

Листинг 3 – Пример объявления класса

```
class Array {
    // ...
}
```

Типы могут быть указаны явно как в листинге 1. Или же они могут быть опущены как тип возвращаемого значения, как в листинге 2. Это возможно потому, что компилятор может вывести некоторые типы на этапе семантического анализа.

Другим важным понятием является сигнатура функции. Она включает в себя тип функции, но кроме того содержит дополнительные сведения, например модификаторы доступа. Важность этого легко продемонтрировать на примере модификатора static, который в таких языкак как java или swift обозначает является ли член номинатива объектом экземпляра или самого класса. От этого зависит способ его использования:

Листинг 4 – Пример вызова статической и нестатической функции

```
// calling static method
Array x = Array.createEmpty()

// calling instance method
Boolean successfully = x.add(42)
```

В случае вызова метода экземпляра size, он вызывается на экземпляре класса, по другому это можно рассмотреть с той точки зрения, что функции size необходим ресивер, то есть экземпляр относительно которого функция проделает свою работу. Возможность переставления ресивера в качестве первого параметра функции называется «универсальный синтаксис вызова функций»(UFCS). Исходя из этих соображений можно положить, что во входной тип функции входит не только тип Integer, но и Array. Подобные рассуждения будут использоваться и в дальнейшем для упрощения рассматриваемого предмета.

Следующая особенность, которую удобно иметь в виду касается рассмотрения функций нефункциональных языков с функциональной точки зрения. Так тип функции, имеющей несколько аргументов, например add из листинга 4, которой может быть записан в виде Array, Integer −> Boolean можно подвергнуть процедуре каррирования, то есть преобразованию во множество вложенных функциональных типов, причём это можно сделать разными способами, в данном случае Array −> (Integer −> Boolean) или Integer −> (Array −>

Boolean). Такой способ рассмотрения имеет место быть и для языков, где функции не являются объектами первого класса. Любой функциональный тип можно представить как стрелку от одного входного и одного выходного типа. Этот факт в дальнейшем покажет что иерархию номинативных типов можно «продлить» и на функциональные типы.

1.2. Системы типов

Языки программировния и системы типов можно разделить по множеству критериев. Для работы важно отметить отличия **номинативного** и **структурного** подхода подтипизаций. Они отличаются в возможности подстановки одного номинативного типа вместо другого. Номинативный подход требует явного указания иерархии наследования, в таком случае объект типа наследника, имеющий функционал родительского типа и возможно дополнительный, может быть использован в том месте, где ожидается вышестоящий тип, то есть родитель. Структурная типизация более свободная в этом плане, в ней достаточно, чтобы у значения типа A, которое подставляется туда, где ожидается тип B был интерфейс взаимодействия, совпадающий с интерфейсом типа B, то есть такие же поля и методы, то есть схожая структура.

Про **подстановку** можно рассуждать и в контексте функций. Тут становиться уместным понятие вариантности, поскольку на тип функции можно взглянуть как на контейнер для типов. В наиболее общем виде функциональный тип это стрелка A -> B, где A и B — какие-то типы. Пусть тип C можно подставить вместо типа B, тогда A -> C можно подставить вместо A -> B, отношение перенеслось с типов на контейнеры, то есть это ковариантность, в таком случае говорится, что функция ковариантна по возвращаемому типу. Для примера, почему такая вариантность логична, можно взять тип Even(чётное число), который можно подставить вместо типа Integer(целое число). Логика в том что любое чётное число является целым. Тогда функцию, которая возращает Even можно подставить туда, где ожидается функция, которая возвращает Integer. С другой стороны, функции можно положить контравариантными по входному типу, ведь, если входной тип ожидаемой функции Even то вместо неё можно передать функцию с входным типом Integer, потому что функция, принимающая целое число, способна принять любое чётное.

Вообще говоря, вариантность функций может быть произвольной, в том числе её может не быть вовсе, то есть тогда будет требование на точное соот-

ветсвие ожидаемого типа и типа подставляемой функции. Или она может быть совсем не нужна, если функции не являются объектами первого класса.

Следующей важной особенностью системы типов является полиморфизм — возможность единообразно обрабатывать разные типы. Array<T> — простой пример, обозначающий параметризованный тип, то есть тип с параметром T, вместо которого может быть подставлен другой тип. Этот механизм очень удобен, например в случаях коллекций, таких как массив, множество и т.д., потому что позволяет унифицировать код, который работает с разными типами. Тема параметрического полиморфизма довольно обширна. Из минимального, что ещё стоит отметить, является механизм ограничений(Constraints), который позволяет накладывать ограничения на типы, которые могут быть подставлены. Например Array<T> where T < Comparable означает что тип, которым можно параметризовать, обязан реализовывать Comparable(сравнимость). В дальнейшего удобно воспользоваться подоходом к рассмотрению механизма параметрического полиморфизма используемым в языке zig [12]. В этом языке есть тип type, который позволяет рассматривать параметризуемые типы, как функции, которые принимают тип в качестве аргумента, для примера pub fn Array(comptime T: type) type

Стоит сделать небольшое отступление, чтобы объяснить зачем это нужно. Можно посмотреть на типы с двух точек зрения. С одной стороны — это может быть конкретный тип, например Array, а с другой общий T. Но если рассматривать тип с точки зрения **подстановки**, то эти тип и дженерик параметр не так сильно и отличаются. Они все определяют множество значений, которые можно подставить на их место. Такое обобщение может быть достаточно спорным, для решаемой задачи оно оказывается весьма удачным.

Главный вывод для данной работы, в том, что вне зависимости от системы типов, небходимо знать как одни типы подставлять на место других, для составных типов это определяется на основе вариантности.

1.3. Терминология

Определение 1. **Декларация** — структурированный фрагмент программы, означающий объявление сущности, изначально определённый в виде целостного блока текста, который в процессе компиляции приобретает семантические свойства такие как тип, область видимости, принадлежность к другим декларациям.

Очевидный пример декларации — объявление функции. Вызов функции в текущем рассмотрении не относится к декларациям, потому что не является объявлением, однако использует декларации, объявленные ранее.

Определение 2. **Тип** — семантическая информация, метка присущая одновременно и декларациям и конкретным объектам, представителям типа, определяет доступный функционал взаимодействия.

Например, функциональный тип определяет тип значений, которые могут быть использованы в качестве аргументов значения функционального типа.

Типы можно разделить на две группы - несоставные и составные.

Определение 3. **Сигнатура** — присущая декларации информация, включающая себя тип и прочие существенные сведения такие как, модификаторы, соотношения с другими декларациями и информацию о дженерик параметрах с их ограничениями.

Определение 4. **Фрагментированный типовой путь** — A ~> B, где слева от ~> стоит входной параметр, а справа выходной, так что существует произвольное колличество функций, композиция которых будет иметь A входным и B выходным. Входной и выходной параметр могут быть как типами, так и фрагметированными типовыми путями.

Определение 5. **Типовой путь** — либо тип, либо фрагментированный типовой путь.

Определение 6. **Фрагментированная типовая сигнатура** — типовой путь с дополнительной информацией в виде ограничений дженериков, наличия параметров по умолчанию и именованных аргументах.

1.4. Формальная постановка задачи

Дано множество деклараций и соотетсвующих им сигнатур, а так же особенности подстановки. Требуется по поисковым запросам являющимся типовыми путями составить ранжированный список кандидатов удовлетворяющим запросу.

Выводы по главе 1

Самым важным принципом в рассматриваемом вопросе является **подстановка**. Комбинация выражений согласуется комбинацией их типов, что в свою очередь определяется возможностью подставить один тип вместо дру-

го. В этих терминах можно рассматривать и дженерик параметры, которые могут иметь ограничения и, опять же, — это ограничения на подстановку.

ГЛАВА 2. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

2.1. Краткий обзор

Вначале идёт описание грамматики запросов и семантика, которая за ней стоит. Затем приводится ряд примеров демонстрирующих нюансы, которые необходимо учесть в рамках решения. Далее идёт описание поискового индекса — ключевого элемента, позволяющего производить поиск. Далее описание процедур добавления новой декларации согласно её сигнатуре и поиска. После этого рассматривается ряд дополнительных вопросов, вроде алгоритмов поиска позиции типа в графе подтипизации, ассимптотической сложности операций, оптимальности данной структуры, а так же возможные улучшения.

2.2. Грамматика запросов

Грамматика представлена в вариации формы Бэкуса — Наура. Непустые строки представляют собой законченные конструкции. Слева от ::= стоит определяемый терм, справа одна возможных его замен. В кавычки заключены терминалы — конечные строки. Терм NAME означает произвольную непробельную строку — идентификатор, состоящую из строчных и заглавных букв и некоторых специальных символов. Терм NO_SPACE означает отсутсвие пробельных символов между стоящих сбоку термов. Во всех прочих местах между термами пробелы могут встречаться в произовльном колличестве. Терм TypeList <DELIMITER> является термом — шаблоном, где на место DELIMITER может быть подставлен произвольный терминал — строка. Это имеет целью удобство задания грамматики и означает лишь то, что списки типов могут быть разделены разными символами — либо запятой, либо амперсандом. Терм EMPTY означает пустую стоку. Вспомогательный символ астериск — $*$ означает произвольной, возможно нулевой, длины последовательность термов, которые записаны в круглых скобках.

После ключевого слово where указываются ограничения на типы перечисленные через запятую.

Стрелки по грамматике получаются правоассоциативными.

Листинг 5 – Грамматика запросов

```
Query ::= Type Constraints

Arrow ::= "->"
Arrow ::= "~>"
```

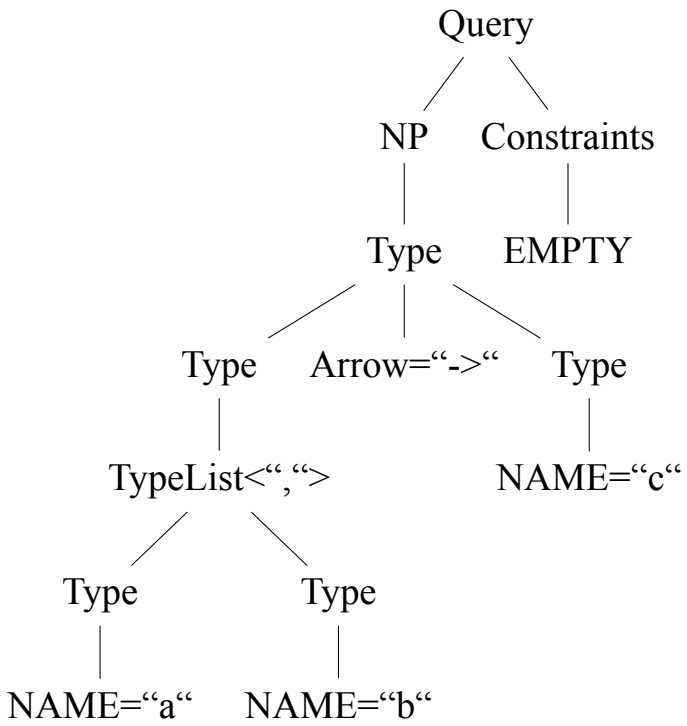
```
Type ::= "!" Type
Type ::= NAME
Type ::= NAME "<" NO_SPACE Type ">"
Type ::= "(" Type ")"
Type ::= TypeList<",">
Type ::= Type Arrow Type

TypeList<DELIMITER> ::= Type (DELIMITER Type)*

Constraints ::= EMPTY
Constraints ::= "where" TypeConstrains ("," TypeConstrains
    )*

TypeConstrains ::= Type "<" TypeList<"&">
```

Отдельно стоит отметить тот факт, что вышестоящие выражения имеют более высокий приоритет применения по отношению к ниже стоящим. Для примера `a, b -> c` представляет собой следующую структру:



Ниже в листингах представлены примеры выражений, которые могут быть распарсены данной грамматикой.

Листинг 6 – Стрелка из a в b

```
Int -> Int
```

Листинг 7 – Стрелка из двух неупорядоченных типов, что соответсвует функциональному типу с точки зрения ООП языков вроде Java

```
Int, String -> Char
```

Листинг 8 – Стрелка между двумя номинативными типами, которые параметризованы дженериками

```
Array<T> -> Collection<T>
```

Листинг 9 – Стрелка, возвращающая функциональный тип, принимающий на вход Int -> T и возвращающая T, который имеет ограничение

```
Int -> ((Int -> T) -> T) where T < ToString
```

Листинг 10 – Другие более сложные примеры

```
A<T> -> B<G> where T < ToString & Equtable<T>, G <
    ToString
A<B<T>> -> (T<A>, T<B>)
A<T> -> B<T> where A<T> < B<T>
A<T> -> Option<B<T>> where T < Equtable
```

Поскольку грамматика задана в несколько вольной форме, её парсинг производится с некоторыми ухищрениями. В общем счёте можно сказать, что он идёт жадно слева направо, но для корректного разбора выражения с запятыми, например A, B -> C, D, которое семантически эквиваленто ‘[A, B] -> [C, D]‘, производится перестановка вершин в уже разобранном дереве. Важно отметить что для разбора не требуется идеально линейный по времени от размера входных данных алгоритм, и реализованного парсера более чем достаточно для решения задачи. Код парсера содержиться в query_parser.zig в исходном коде [3].

2.2.1. Семантика грамматики

Семантика по большей части соотноситься с общепринятой. Слева от стрелки входной тип, справа выходной. Круглые скобки используются для обозначения приоритета. Типы, перечисленные через запятую в круглых скобках обозначают кортеж — упорядоченный набор типов. Типы, перечисленные через запятую, но не обрамлённые в круглые скобки — неупорядоченное множество типов, ближайщая аналогия — входной тип функции в ООП мире, где переупорядочивание аргументов не меняет семантики. Такие неупорядоченные

множества типов для удобства можно обозначать в квадратных скобках. Угловые скобки означают параметризацию типов. Прямая аналогия — дженерик параметры. Ограничения задают отношения между типами. Восклицательный знак перед типом означает что последний в контекстке этой декларации имеет значение по умолчанию. Например параметр функции имеет аргумент по умолчанию, то есть его можно не указывать при вызове.

Для упрощения, в работе делается допущение, что дженерики представлены заглавными одиночными буквами. Это позволяет не вводить их заранее, как это часто делается при объявлении функций и номинативов, и не привносит неудобство, поскольку реальные номинативы в стандартных библиотеках и пользовательском коде чаще имеют осмыленные длинные имена, во всяком случае длиннее одного символа. От этого ограничения не сложно избавиться, но для начала проще оставить это допущение.

Стоит отдельно отметить, что далеко не все запросы, которые могут быть разобранны грамматикой имеют смысл в целевом языке. Однако это не является первостепенной проблемой. Поскольку при использовании таких запросов при поиске просто не найдётся подходящих кандидатов. Это вполне соответсвует ожидаемому поведению.

Данная грамматика может быть переиспользована и для цели передачи информации о декларациях и соответсвующим им типов из целевого языка. Требуется лишь сформировать корректную сигнатуру согласно данной грамматике. Этого будет достаточно, чтобы использовать её в структуре поиска, описанного в следующем разделе.

2.3. Нюансы задачи

В первом приближении задача поиска деклараций по сигнатуре может показаться как весьма простой, так и довольно сложной. В ходе работы было выясненно огромное колличество нюансов, которые не позволяют подходить к решению, как к простому алгоритму на строках. Во-первых, очевидно, нужно уметь находить декларацию, сигнатура которой в точности соответсвует запросу. Например, функция `foo : String -> Int` должна находиться по запросу `String -> Int`. Если же тип `IntEven` является подтипом `Int`, и имеется функция `boo : String -> IntEven`, то она так же должна находиться по запросу выше. Общими словами можно сказать, что тип функции `boo` подставим на место где ожидается тип, указанный в запросе. Подставимость функции определяется вариант-

ностью, которую учитывать на любом уровне вложенности. Две описанные выше функции так же являются валидными кандидатами для запроса T -> G, где T и G это дженерики, то есть позиции на место которых можно подставить любой прочий тип. В том числе под запрос T -> G валидным кандидатом будет функция (String, Int) -> (Int -> Int). Кроме прочего, важно уметь отличать функции вида x : T -> T и y : T -> G, поскольку у второй на входной и выходной позиции стоят два разных дженерика, а у первой один и тот же. Следующий пример касается функций высшего порядка: (A -> B) -> C и A -> B -> C — это два разных запроса, и кандидат, соответсвующий типу одной, не должен подходить другой. Далее под запрос вида T -> Int может подойти кандидат Array<Int> -> Int. Ситуация становиться ещё сложнее, когда встаёт необходимость учёта ограничений. Так функция z : Array<T> where Array<T> < Printable может быть кандидатом запроса R where R < Printable. Есть множество более сложных случаев. Отдельный вопрос касается поддержки рекурсивных ограничений T where T < Equtable<T>. Кроме вышеперечисленных проблем, связанных с подтипизацией, необходимо учесть альфа-эквивалетность дженерик параметров, то есть возможности замены их имени.

Перечисленные выше частные случае показывают необходимость учёта множество особенностей. Очевидно, что наивным перебором не обойтись, и кроме того он может оказаться слишком медленным. Описанное далее решение, основанное на идеях префиксного дерева, позволяет решать поставленную задачу, не совершая шаги, заведомо приводящие к некорректным кандидатам.

2.4. Поисковый индекс

Структура представляет собой граф. Он образован двумя видами вершин — Node и TypeNode и связями между ними. Ориентированные рёбра между двумя вершинами TypeNode представляют собой отношение подтипизации между типами. Ребра, исходящие из TypeNode в Node, представляют собой переходы, которые означают, что к типовому пути собранному до данного момента, добавляется теперь и продолжение, представленное TypeNode'ой. У вершин типа Node имеется множество деклараций, которые имеют сигнатуру, соответствующую пути, пройденному до неё. Вершинам типа Node принадлежит граф отношений между TypeNode (граф подтипизации). Часть вершин этого графа представлеют собой конкретные типы, а другая часть — типы, не

представленные в языке явно (к таким можно отнести дженерики, на которые наложены ограничения. У них во всяком случае часто нет имён, а значение в системе типов языка является не до конца определённым). К другим видам TypeNode можно отнести универсальный тип и скобки (они являются по большей части вспомогательным, и имеют в общем тоже назначение, что и в строковом представлении типов). Вершины TypeNode связанные отношением подтипизации могут принадлежать разным Node'ам, такое получается например когда дженерик имеет функциональное ограничение. Например `T where T < (Int -> String)`.

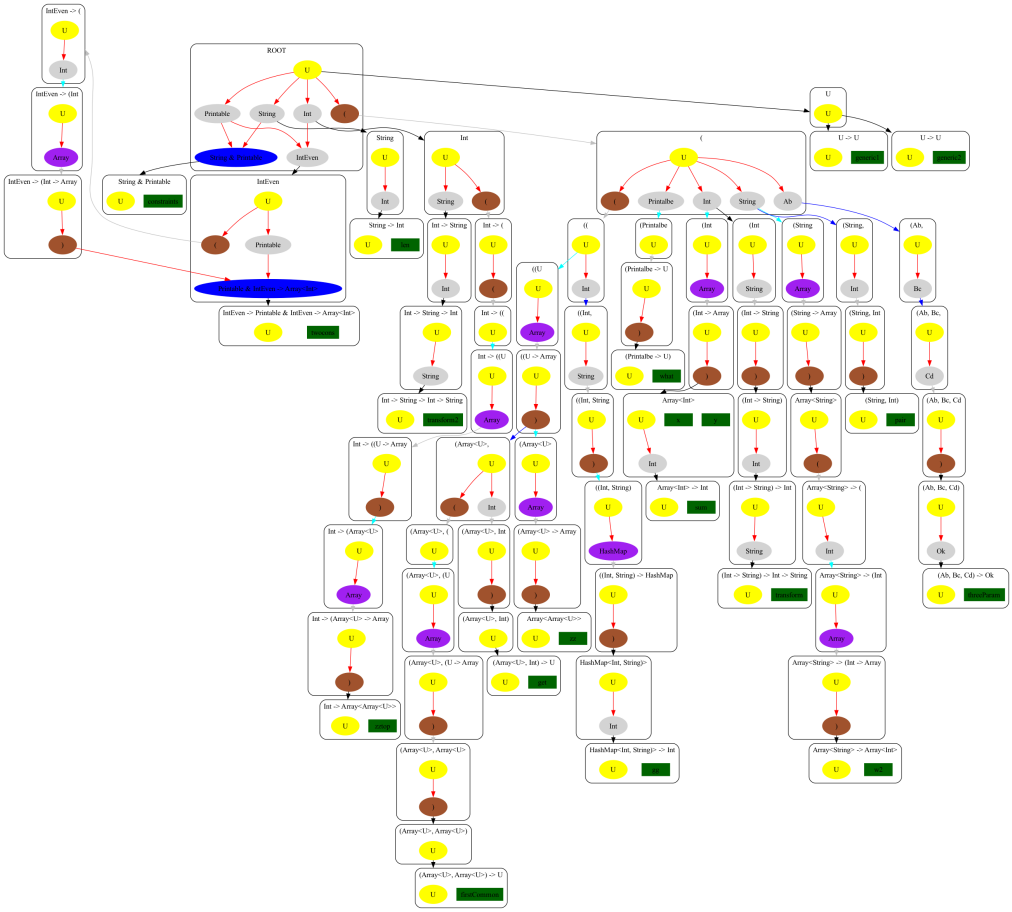


Рисунок 1 – Пример поискового индекса

Листинг 11 – Множество деклараций на основании которых построен поисковый индекс на рисунке 1

```
len : String -> Int
sum : Array<Int> -> Int
transform : (Int -> String) -> Int -> String
transform2 : Int -> String -> Int -> String
generic1 : T -> T
```

```
generic2 : T -> G
constraints : T where T < Printable & String
twocons : IntEven -> T where T < Printable & Array<Int>
what : G<T> where T < Printalbe , G < Printable
w2 : Array<String> -> Array<Int>
zz : Array<Array<T>>
zztop : Int -> Array<Array<T>>
gg : HashMap<Int , String> -> Int
x : Array<Int>
y : Array<Int>
pair : (String , Int)
get : Int , Array<T> -> T
firstCommon: Array<T>, Array<T> -> T
threeParam: Ab, Bc, Cd -> Ok
```

Стрелки между вершинами:

- Красные — отношение подстановки
- От TypeNode к Node:
  - Черные — функциональные
  - Синие — запятые кортежей
  - Сине-зеленые — стрелки между дженериками и содержащими их номинативами
  - Серые — от или к служебной вершине

Цвета TypeNode:

- Жёлтый — универсальный тип
- Серый — номинатив
- Коричневые — скобки
- Синие — синтетические ноды, для представления типов, которые заданы ограничениями
- Фиолетовые — бывшие номинативы с дженериками (Чтобы отличать Array<T> от Array)

Поисковой индекс внешне похож на дерево, особенно, если смотреть на него как на множество Node. Эта особенность связана с тем, что изначальная его

идея основана на префиксом дереве. А TypeNode'ы решают большую часть нюансов описанных ранее, которые связаны с определение подтипизации.

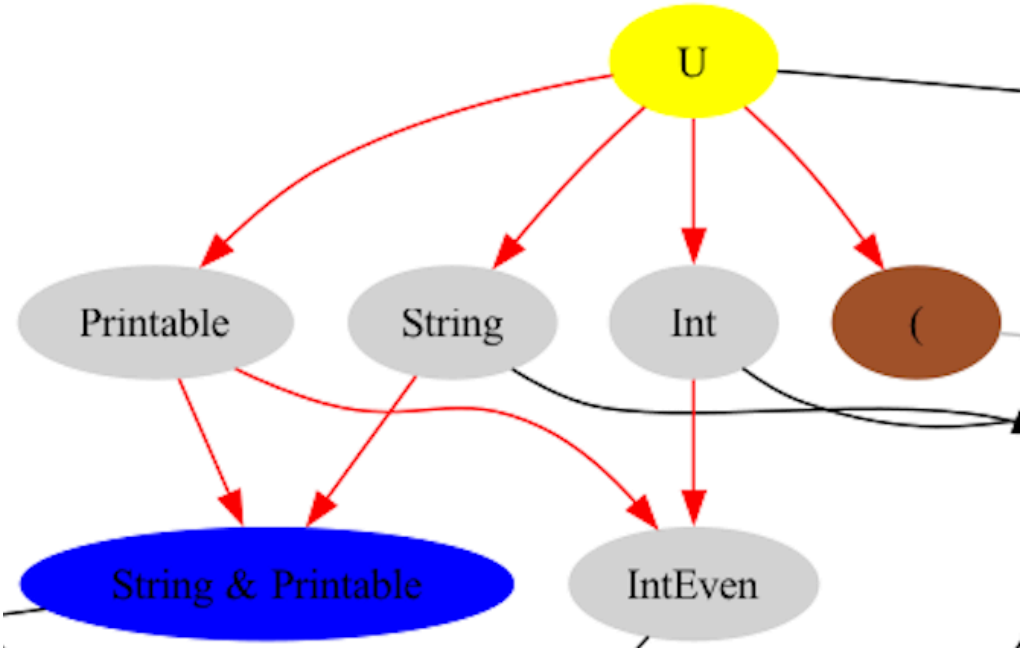


Рисунок 2 – Граф подтипизации

Каждая новая Node состоит из графа подтипизации, содержащей только один универсальный тип и пустого множество деклараций, которые закончились в этой ноде. При построении индекса в граф подтипизации добавляются новые типы и связи между ними.

Граф подтипизации это частично упорядоченное множество построенное между типами, которые логически принажлежат одной Node, в нём также отражены и ограничения на типы. Отношения подтипизации могут связывать любые два типа.

Листинг 12 – Множество деклараций на основании которых построен поисковый индекс на рисунке 3

```
T_AnBn: T where T < An & Bn
T_AnCn: T where T < Cn & An
T_CnDn: T where T < Cn & Dn
AnDn : AnDn
T_AnDn : T where T < An & Dn
AnBnCn: AnBnCn
T_AnBnCn: T where T < An & Bn & Cn
T_CnT_AnDn : T where T < AnDn & Cn
T_AnCnDn : T where T < An & Cn & Dn
```

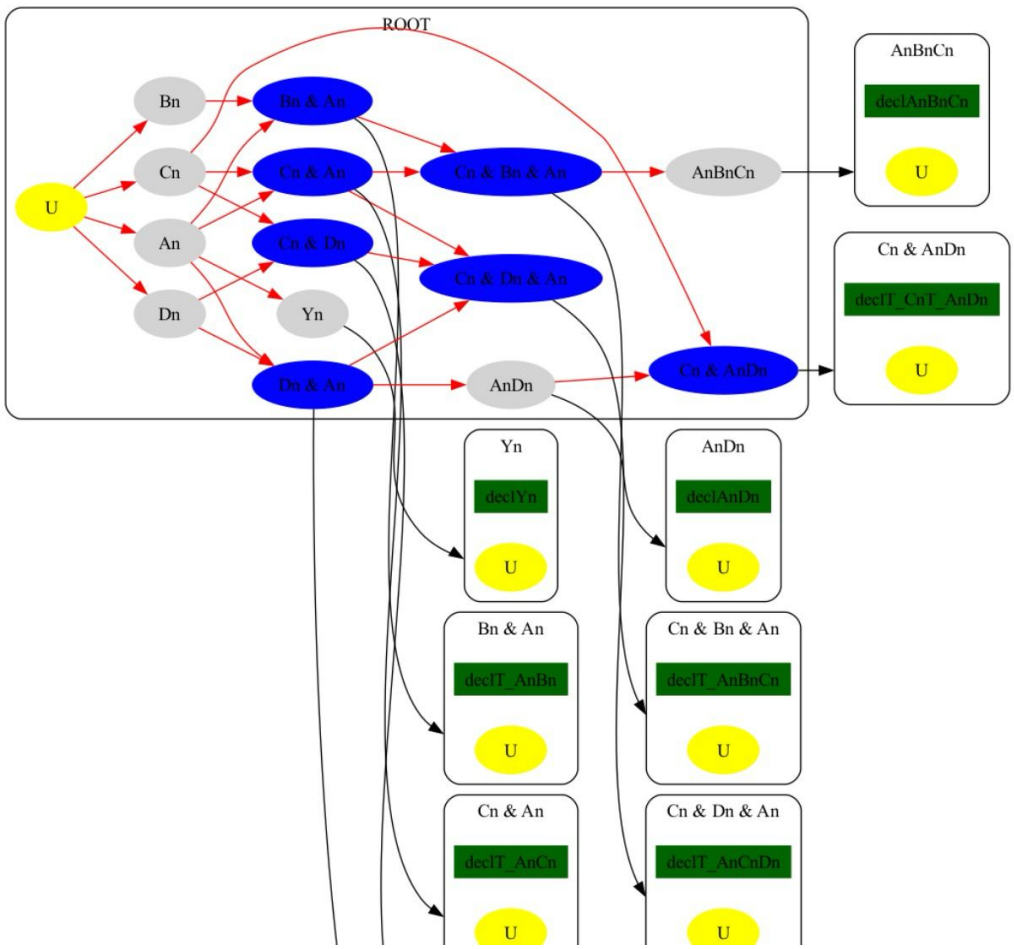


Рисунок 3 – Более сложный пример графа подтипизации

Листинг 13 – Отношение подтипизации для типов с рисунка 3

```
AnBnCn < An
AnBnCn < Bn
AnBnCn < Cn
AnDn < An
AnDn < Dn
Yn < An
```

У каждой TypeNode имеется множество переходов в Node’ы, каждому переходу соответсвует пара, первым элементом которой является целевая Node, а вторая — указатель на TypeNode, которая представляет дженерик, встречавшийся ранее и имеющий такие же ограничения или null, если эта TypeNode не является дженериком. В большем числе случаев будет иметься лишь одна пара, второе значение которой является null указателем, то есть означающим, что этот тип встречается впервые. Это верно для абсолютно всех конкрентых типов, поскольку они всегда означают одно и то же в отличие от дженериков. Для демонстрации этого — тип T -> G where T < Int & G < Int имеет

два дженерика, которые, хоть и имеют общее ограничение, могут быть совершенно разными типами в отличие от типа Int -> Int, где и первый и второй Int указывают на одно и то же.

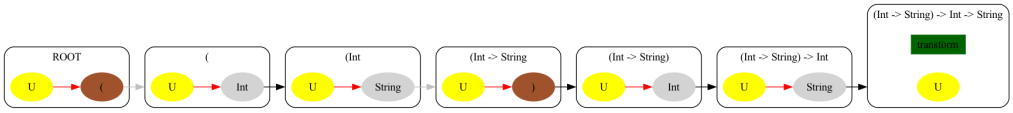


Рисунок 4 – Дерево построенное по декларации с типом (Int -> String) -> Int -> String

Весь индекс является «плоским». То есть функции высшего порядка хранятся и обрабатываются вне зависимости от вложенности. Функциональный тип является подтипом универасального, что так же отражено в графе подтипизации представленного TypeNode'ами. Однако важно учитывать, что и между функциональными типами могут быть подстановки, от чего сопостовлять функциональный тип целеком одной TypeNode недостаточно. Решением этого вопроса является использование двух специальных TypeNode "(" и ")" . Открывающаяся скобка в графе подставиммости внутри Node представлена TypeNode. Однако семантически это значит что подставима не одна ли открывающаяся скобка, а тип, который представлен между открывающей и соответствующей закрывающей скобками. Это имеет прямую аналогию с тем какой смысл скобки имеют в строках.

Номнативы с дженериками в некотором роде подобны функциональным типам. Они точно так же подставляемы на позицию универсального типа, и у них точно так же могут быть отношения подтипизации, основанные на вариантности и подставимости типов аргументов. Пример, хоть может и не релевантный к большей части популярных совеременных языков — Array<Int -> T > where T < Printable , тем не менее ясно показывает что тут не обойтись одной TypeNode, в связи с тем что подставимость может зависеть от внутренностей. Решением данного вопроса является приведение дженерика к виду функционального типа. Для примера Array<T> преобразуется в T -> Array∗. Астериск носит вспомогательный характер, означающий, что это не то же самое, что и тип, ничем не параметризованный и имеющий такое имя (Array), и больше не несущий никакой смысловой нагрузки. Такой подход к трансформации дженерик параметризованных типов имеет приемущество, в первую очередь связанную с тем, что это автоматически учитывает вариантность и встраивается в текущую

концепцию поискового индекса, а добавление и поиск происходит аналогично тому, как это делается для функционального типа.

Поисковой индекс, организованный подобным образом, имеет приемуществом ещё и возможность распараллеливать вычисления.

Важной возможностью такой структуры является способность устанавливать отношение подтипизации и для функциональных типов, которыми в частности представлены дженерики.

2.4.1. Добавление декларации в поисковый индекс

Операция добавления происходит итеративно, на каждом шаге имеется Node и тип, для которого необходимо вернуть продолжение, то есть соответсвующую TypeNode. Для разных типов этот процесс различен.

Для функции создаётся служебная TypeNode, соответсвующая открывающейся скобке. По ней делается переход в следующую Node. Далее рекурсивно вставляется входной тип функции. Затем из продолжения переход в дальнейшую Node, в которую рекурсивно вставляется выходной тип функции. Далее по аналогии служебная TypeNode, соответствующая закрывающейся скобке. Стрелка-переход между входным и выходным типами является функциональной стрелкой, а стрелки после открывающейся скобки и перед закрывающейся — служебные.

Для кортежа также вставляются служебные TypeNode для скобок, между которыми вставляются типы, которые являются частями кортежа. Стрелки между ними соответсвуют запятым.

Для номинатива, если он не параметризован, то просто добавляется в текущую Node. А если у него есть дженерик, то эта конструкция разварачивается в функцию, которая на входе имеет тип соответсвующий дженерику, а на выходе номинатив без параметров. Эту функция вставляется почти так же как обычная функция высшего порядка, за ислючением что между входным и выходным типом функциональная стрелка имеет особую пометку, сообщающую о том, что изначально это был параметризованный номинатив.

Подробности принципов работы лучше смотреть непосредственно в исходном коде [3].

2.4.1.1. Добавление типа в граф подтипизации

Добавление типа в граф подставляемости отличается для конкретных типов, то есть представленных в языке, и типов синтетических, то есть таких, которые определяются ограничениями.

Для конкретных типов алгоритм вставки следующий. Создаётся множество старших типов, которые будут динамически обновляться. Изначально в него добавляется универсальный тип, поскольку он является старшим к любому прочему. Затем проводиться операция «проталкивания» вниз. Для типов из множества старших перебирается множество непосредственных детей, для которых проверяется что этот ребёнок больше вставляемого типа (является предком), если это так, то он добавляется в множество страших типов. Если после проведения этой процедуры для конкретного старшего типа не один из его непосредственных детей не оказался старшим по отношению к вставляемому, то вставляемый становиться дочерним по отношение к рассматриваемому старшему. Если в таком случае вставляемый становиться старшим по отношению к какому-либо ребёнку старшего типа, то он переподвешивается как ребёнок к вставляемому. Таким образом новая TypeNode, соответсвующая вставляемому типу, становиться на своё место. Далее для прочих типов необходимо проверить не являются ли они дочерними к добавляемому. Эту процедура можно значительно ускорить заранее сообщая информацию ближайщих старших и младших типах.

Для TypeNode, представляющей синтетический тип так же инициализируется множестсво старших, которые изначально добавляются заданные ограничения. Процедура вставки представляет собой нахождение нижней грани множества старших TypeNode в полной решётке [2]. Если для двух TypeNode из множества существует нижняя граница, то она заменяется на эту соответсвующую TypeNode, представляющую эту границу. Если нижняя граница ещё отсутствует в ЧУМ [10], то она добавляется. Все эти операции, конечно же, учитывают необходимость переподвешивания вершин.

Стоит отметить факт, что функциональные типы являются полноценными участниками этой схемы.

Подробности того как это происходит лучше смотреть в исходном коде [3].

2.4.2. Поиск с учётом вариантности

Поиск с учётом вариантности напоминает поиск TypeNode по соответсвующему типу. Отличие состоит в том, что при последовательном спуске при поиске точного типа, переход делается не в одну Node, следующую из текущей TypeNode, а в несколько Node из нескольких TypeNode. Соответсвующие TypeNode берутся как старшие или младшие к TypeNode представляющей точное совпадаение, что определяется во вариантностью, принципы вычисления которой будут описаны ниже. Таким образом получается не одна подходящая TypeNode, а множество, для каждой из которых в дальнешем будет проведена процедура поиска с соответсвующей декларацией.

2.4.2.1. Вычисление вариантности

Тип может быть ковариантным (+), контравариантным (-), инвариантным (0), бивариантным (∞), в зависимости от своего положения — входная позиция функции, выходная позиция функции, дженерик параметр, тип в кортеже. Наиболее интутивными можно считать следующую конфигурацию — функция контравариантна по входному типу, функция ковариатна по выходному типу, номинатив с дженериком инвариатен, кортеж ковариантен по каждому типу. Можно ещё сказать, что тип по отношению сам к себе является ковариантным. Изначально тип ковариантен, далее, если он является составным, то для его частей вычисляется перемножение вариантностей на основании конфигурации и дошедшего до данного места значения вариантности. Перемножение происходит по таблице на рисунке 5.

	0	-	+	∞
0	0	0	0	0
-	0	+	-	∞
+	0	-	+	∞
∞	0	∞	∞	∞

Рисунок 5 – Таблица перемножения вариантности

2.4.3. Конструирование выражения

Конструирование выражений можно рассматривать как композицию функциональных типов. Выходной тип первого типаA –> B должен совпадать с входным второго B –> C. Проверку равенства двух типов можно осуществить одновременно совершая одинаковые шаги в двух местах. Кроме того можно

совершать шаги с учётом вариантности, так чтобы выходной тип первого типа был подтипом второго. То есть алгоритм композии состоит в том чтобы сначала удовлетворить входной тип первого функционального типа, а затем начиная с этого места совершать соответсвующие шаги из корня дерева. Нюансы касающие параметров со значениями по умолчанию и другие так же естесвенно встраиваются в этот алгоритм.

По другому на алгоритм можно посмотреть как на зеркальный поиск. Изначально находится множество вершин, тип пути которых соотвествует входному типу первой функции. Далее для каждой такой вершины A и корня дерева делаются зекральные шаги (то есть одинаковые вплоть до совпадения дженериков), и в тех Node'ах, которые входят в путь из A, где имеются декларации, эти декларации запоминаются как внутренняя часть композиции. А по зеркальным вершинам которые соответвуют путям вышедших из корня в дальнейшем делается проход соответсвующий выходному типу, и декларации которые встречаются в достигнутой вершине становятся наружней частью композии функций. Стоит обратить внимание что внутренний зеркальный проход можно делаеть не в точности, а в соответсвии с вариантностью. Итогу получается множество пар деклараций, которые в композии будут иметь заднными входной и выходной типы.

Выводы по главе 2

Описанные алгоритмы позволяют структурно работать с типами, учитывая концепции систем типов, такие как вариантность, наследование, функции высшего порядка, номинативы с дженериками и прочее.

ГЛАВА 3. АНАЛИЗ РЕШЕНИЯ

Данная глава содержит технический обзор полученного решения и описание методологий проверки качества.

3.1. Обзор решения

Предыдущая глава содержит большую часть информации о теоретическом аспекте решения, в данной секции приводится общий обзор компонент и рассматривается ряд технических моментов.

Для начала стоит подробнее рассказать о выборе инструментов, в частности языка разработки. Предпочтение было отдано языку программирования zig [4], который в первую очередь расчитан на разработку системного програмного обеспечения, с ручным управлением памятью. Выбор основан необходимостью написания быстрого и ресурсоёмкого приложения, что имеет две причины. Во-первых подобная утилита относиться к классу инструментов разработчика, отчего должна обеспечивать быстродействие и бережливое отношение к ресурсам хостящей машины. Во-вторых именно ручное управление памятью в совокупности со знанием алгоритмов и принципов устройства компьютеров, позволяет получать самые эффективные решения. Исходя из принципов организации структуры данных и алгоритмов можно сделать вывод, известный всем кто когда-либо писал на ассемблере, о том что основные задержки связаны именно с доступом к памяти при условии, что прочие операции выполняются за константное время. Это явление принято называть «стеной памяти» (eng: «Memory Wall» [7]). Предпочтение языку zig было отдано и по некоторым другим соображениям, во-первых это механизм вычислений во время компиляции, который можно применить в качестве дальнейших оптимизаций. Это логически соответствует тому что информация о декларациях целевого языка может быть известна уже на этапе компиляции. То есть поисковой индекс можно ещё более оптимально разместить в памяти. Кроме того, zig — открытый язык с идеологией устойчивого развития и может компилировать код под самые разные архитектуры. В любом случае, это может быть делом вкуса, но факт в том что полученное решение весьма эффективно.

Проект разбит на несколько компонент: парсер, поисковой движок, визуализатор, кэш — граф подставимости и драйвер. Все это компоненты используют структуры, которые можно разделить на два класса — структуры описания информации о декларациях и типах: Declaration, Nominative, Function, List,

Type, Constraint и структуры для описания фрагментов поискового индекса: Node, TypeNode, Following(ребро и семантическое информация о нём).

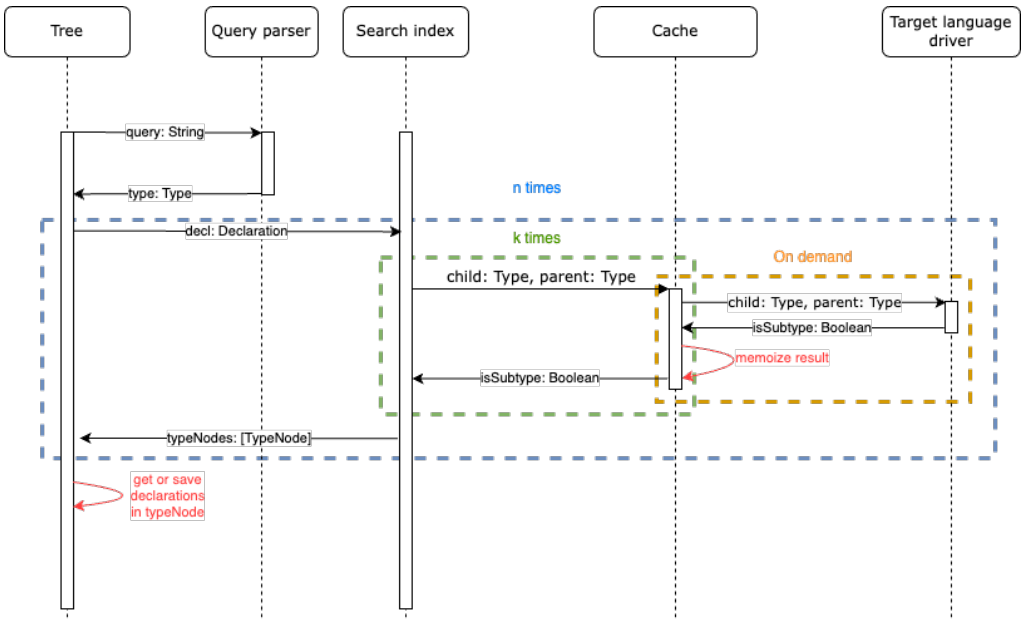


Рисунок 6 – Схема перемещения данных между основными компонентами

Парсер не является критическим по производительности местом. Его реализация не соответсвует типам LL-(k) или LR. Однако он справляется с основной задачей — разбором типовго выражения.

Поисковой движок естественным образом использует поисковый индекс. Он представлен рядом методов структур Tree, Node и TypeNode. Для спуска по дереву, от его вершины к корню, то есть последовательному удовлетворению поискового запроса, в методы передаются указатели на соотвутсвующие вершины и тип, для которого необходимо совершит поиск.

Визуализация поискового индекса осуществляется при помощи утилиты dot из пакета graphviz [1]. По имеющейся информации о вершинах и рёбрах формируется текстовый файл, который и служит входной информацией для визуализатора.

Кэш, необходимый для того чтобы сохранять информацию о подставимости типов, а не обращаться к драйверу целевого языка, представлен точно такой же структурой, что и поисковый индекс. В отличие от Node, в которой только хранятся типы, по которым можно делать дальнейший переход, в кэше постепенно накапливается информация вообще о всех типах и их отношениях подтипизации. Стоит заметить что алгоритмы проверки подтипизации в кеше несколько отличаются от проверки подтипизации при помощи кеша, во втором

случае проверка что один тип является подтипом другого осуществляется через проверку вхождения его в множество верхних граней дочернего. А проверка при отсутствии одного из типов к кэше делается через обращение к оракулу драйвера целевого языка, то есть функции принимающей два типа о определяющей связанны ли они отношением подтипизации.

Стоит пару слов сказать о возможности распаралеливать поиск и другие операции с деревом. Это стало возможно благодоря древовидному представлению поискового индекса и итеративности алгоритма поиска. Что может иметь смысл как для возможности одновременного поиска ответов на несколько запросов в одно и то же время, так и для ускорения поиска с учётом вариантности или конструирования выражений.

3.2. Сравнение с существующими решениями

Аналоги, к которым относятся Hoogle [8], Cloogle [13] и Inkuire [**inkuire**], в первую очередь заточены под конкретный язык, то есть решают более специфичную задачу и, как будет показано ниже, используют более наивные подходы, а кроме того они не способны конструировать выражения, состоящие их нескольких деклараций.

3.2.1. Hoogle

Hoogle, пожалуй, самое ранее из решений, которое решает задачу поиска деклараций, как обычный поиск по имени, который выходит за рамки рассмотрения данной работы, так и поиск по типу. Отличительной особенностью является то, что в поисковую выдачу могут входить декларации, не точно соответвующие запросу. Кроме того, hoogle не имеет механизма, позволяющего учитывать наследование и ограничения в общем виде.

Поиск происходит в два этапа [9]. Сначала для заданного запроса считается «отпечаток» , в котором учитывается арность, колличество конструкторов и три самых редких имени в функции. По этому отпечатку находится 100 кандидатов, которые попадают на вторую фазу. Во второй фазе идёт отсеивание невалидных кандидатов, которое возможно благодаря тому, что можно проводить более детальное сравние на уже меньшей выборке. Как показывает практика, выбранный подход оказался весьма успешным для haskell. Этому же способствовала система ранжирования кандидатов с весами, полученными путём голосования. К минусам можно отнести то, что это решение весьма

специфичное для целевого языка и к тому же не позволяет конструировать выражения.

3.2.2. Cloogle

Cloogle — решение для языка Clean [11] использует несколько другой подход. Поисковый индекс представляет из себя дерево, спуск по которому соответсвует уточнению типа. Так для родительской вершины a −> Int дочерними могут быть Int −> Int или String −> Int. Этот достаточно интуитивный подход, но он так же не позволяет работать с множеством нюансов систем типов, такими как наследование, механизм ограничений и вариантностью (например дженериков [5]).

Camil Staps (автор cloogle) на вопрос о сравнении hoogle и cloogle ответил: «The algorithm is crucially different from Hoogle's, which does not rely on unification.» (перевод: Этот алгоритм существенно отличается от алгоритма hoogle, который не опирается на унификацию.)

3.2.3. Inkuire

Наиболее близким с точки зрения определения системы типов является Inkuire, реализованный для поиска в языках Scala и Kotlin. Стоит обратить внимания на язык запросов, который может показаться излишне загруженным, для примера <K, V> Map<K, V>.(K)−> V? или (String) −> Int или Int .() −> String или <T> List<T>.((T) −> Boolean) −> Pair<List<T>, List<T>>. Подобная запись требует точного знания сигнатур деклараций, необходимости явной записи дженерик параметров и лишних скобок. Сложность формирования поискового запроса не такая маленькая проблема, как кажется на первый взгляд, потому что во-первых легче ошибиться, а во-вторых требуется гораздо больше умственных усилий для формирования запроса, например нужно явно определять является ли один из входных типов ресивером или же обычным параметром. В случае, если ресивер является номинативом с дженерик параметром, как в последнем примере, то запрос может стать излишне перегруженным.

В отличие от предыдущих аналогов в Inkjure используется более гибкий подход, который позволяет учитывать особенности ООП языков, такие как наследование и вариантность. Он также использую концептуальную модель графа подстановки.

К сожалению, у Inkjure нет документации или сколь угодно поверхностного обзора устройства поискового движка. Единственное, о чём авторы рас-

сказали в видео презентации [6] — то, что утилита способна учитывать вариантность и наследование. В Inkuire собирается база данных деклараций для которых сохраняется множество практически вся известная информация, а сам поиск происходит как-бы через запросы к этой базе данных.

3.2.4. Итоги сравнения

Существующие аналоги используют специфичные подходы к решению задачи, в то время как предложенное в работе опирается на фундаментльные прицины систем типов и даёт более общий механизм, позволяющий учитывать вариантность, наследование, ограничения в наиболее общем виде. Кроме того, в описанное решение естественным образом встраивается концепция конструирования выражений, а визуализация показывает насколько данный подход естественен и удобен для работы с типами. Быстродействие поиска декларации по предложенному поисковому индексу обеспечивается не за счёт приблизительного поиска, а за счёт того, что на каждом шаге не делается лишних действий, то есть никогда не рассматриваются заранее невалидные кандидаты.

3.3. Валидация решения

Данный раздел описывает каким образом проверяется качество работы каждого из компонентов по отдельности и всей утилиты в целом.

3.3.1. Тестирование парсера запросов

Для тестирования парсера запросов лучше всего подходит самое обычное юнит-тестирование, то есть ряд тестов, где на вход в строковой форме задан поисковой запрос, а для разобранного результата совершаются проверки на соответсвие с ожидаемым. Код тестов представлен в query.zig. Тесты проверяют во-первых что правильно разбираются конструкции такие как — номинативы, функциональные типы, кортежи, номинативы с дженериками, функции высшего порядка. Так же имеются тесты проверяющие что разобранные типы, имеющие одинаковые имена, совпадают вплоть до указателей на соответсвующие структуры. Например, в Array<T> -> Iterable<T> первое и второе вхождение T должны указывать на одну и ту же структуру, представляющую соответсвующий элемент. Кроме этого, проверяется, что правильно разбираются ограничения, указанные после ключевого слова where. Тестируется, что эти ограничения присоединяются к соответсвующим типам.

3.3.2. Тестирование поискового индекса

Тестирование поискового движка, как отдельного компонента, состоит из проверок на добавление деклараций по соответствующим им типам, проверок на точный поиск, на поиск с учётом заданной вариатности, на правильное размещение вложенных типов в графе подставляемости. Тесты приведены в файлах папки src/engine/tests. Этими тестами проверяются все известные случаи и нюансы. Часть тестов нацеленны на проверку корректность построения графа подтипизации.

3.3.3. Тестирование визуализатора

Тестирование визуализатора состоит в основном из проверки корректности названий Node пути пройденному до этой вершины, и TypeNode на соответсвие типам. Также ряд тестов проверяет, что формируемый текстовый файл, который будет использован на вход утилиты dot, содержит правильную информацию.

3.3.4. Общие тесты

Кроме специфичных вышеперечисленных тестов, реализован ряд общих проверок, в которых проверяется, что в процессе работы не происходит утечек памяти, и не происходит лишних аллокаций, то есть указатели на структуру совпадают на одном и на другом концах программы.

3.3.5. Проверка качества утилиты в целом

Поскольку утилита направлена на решение весьма практичной задачи, важно проверять её работоспособность в реальных условиях. Схема проверки весьма простая и опирается на принцип — если код компилируется, то он корректный.

И так, после процедуры поиска имеется запрос (тип, кандидаты для которого ищутся) и найденные декларации. Все остальные декларации не являются подходящими под этот тип. Поэтому схема проверки следующая — генерируем код, состоящий из переменной с типом соотетсвующей декларации, и присваивания этой переменной в другую переменную тип которой соответсвует типу запроса. Такой код должен компилироваться для всех найденных деклараций и выдавать ошибку компиляции для прочих. Такой способ проверки может быть реализован во многих языках. Ниже приведён пример для языка

kotlin. Проверка того, что кандидат подходит под запрос происходит во второй строчке функции main. Это как раз соответсвует тому, что тип декларации подставим вместо типа запроса.

Листинг 14 – Схема валидации решения для языка kotlin

```
fun <T> foo(x: Boolean): List<T> = throw
    NotImplementedError()

fun main() {
    val candidate: (Boolean) -> List<Int> = ::foo
    val query: (Boolean) -> Collection<Int> = candidate
}
```

В листинге 15 приведён пример кода, который не скомпилируется, потому что тип (Boolean) -> List<Int> нельзя использовать там, где ожидается (Boolean) -> Collection<Boolean>. Компилятор напечатает ошибку вида «Type mismatch: inferred type is (Boolean) -> List<Int> but (Boolean) -> Collection<Boolean> was expected».

Листинг 15 – Пример кода, который не скомпилируется

```
fun <T> foo(x: Boolean): List<T> = throw
    NotImplementedError()

fun main() {
    val candidate: (Boolean) -> List<Int> = ::foo
    val query: (Boolean) -> Collection<Boolean> =
        candidate
}
```

Надёжность данного способа валидации решения упирается только в правильность работы компилятора. Хотя сама схема проверки и кажется сложной с вычислительной точки зрения, поскольку для каждого поискового запроса нужно совершить столько компиляций, сколько всего есть деклараций, её, однако, можно ускорить, если добавить предварительную проверку арности типов и другие проверки, которые исключат заранее невалидных кандидатов.

Идеальным результатом может считаться, если все ожидаемо компилирующиеся программы скомпилируются, а прочие нет. Однако, допустимо, что могут возникнуть крайние случаи, которые не будут корректными. Поэтому можно замерить процент корректных запусков компилятора, где результат сов-

падает с ожидаемым ко всему колличеству запусков. Это назовём процентным показателелем качества. Также можно систематизировать случаи корректной и некорректной работы утилиты и проанализировать полученные результаты.

Качественная реализация драйвера написанна только для непубличного языка. Утилита не покрывает лишь часть случаев, которые однако нельзя подробнее описать. Пример демострации возможностей привдён в исходном коде на примере искусственной системы типов, которая однако является весьма приближенной к настоящей. Наиболее долгой операцией является построение поискового индекса, скорость поиска и конструирования выражаения в общем случае зависят от колличества деклараций в индексе и особенностей подтипизации, однако для примеров на которых производилось тестирование в процессе написания, операции поиска и конструирования выражения выполнялись быстрее 1 миллисекунды. По памяти тестовый пример вмещается в 1 мб. В целом данную утилиту можно применять и в реальной жизни, это однако требует ряда настроек специфичных для ситуации.

3.3.6. Сравнение с конкурентами

Из описанных аналогов наиболее подходящим для сравнения является Inkuire. Основная причина в том, что он учитывает отношения подтипизации и вариантность. Количественное сравнение проводилось по 2-м критериям: средней скорости ответа на запрос и процентному показателю качества на выборке из 100 случайных типов. Результаты приведены в папке comparision исходного кода.

Затрудняющим для сравнения принципов устройства является использование весьма разных языков реализации. Inkuire написанный на высокоуровневом языке scala, поддерживающим весьма много абстракций, в то время, как предложенное решение написанно на языке zig, изначально заточенным под системную разработку.

Хотя оба решения учитывают подтипизацию, способы их представления несколько отличаются. В предложенном решении для каждой отдельной Node существует свой граф подставимости, что позволяет значительно быстрее делать поиск с учётом вариантности и наследования. Кроме того, идея префиксного дерева, которое позволяет получать наглядную визуализацию деклараций в пространстве типов, является уникальной в предложенном решении. Inkuire не позволяет комбинировать отдельные декларации в выражения, в то время

как это естественным образом возможно в текущем решенении. Следующее отличие в способе введения дженериков. В Inkuire это делается явно, а не на основании правила о том, что джененрик это одиночная заглавная буква. Ещё одним отличием Inkuire является выделение ресивера, что с одной стороны усложняет синтаксис запросов, но с другой даёт дополнительную точность, польза которой несколько неоднозначна, поскольку для нахождения желаемой декларации будет необходимо воспользоваться поиском два раза, в то время как в предложенном решении полноценная поисковая выдача будет получена по одному запросу.

Выводы по главе 3

Решение состоит из отдельных весьма естественных компонент. Если рассматривать язык как первостепенное средство разработки, то выбор zig имел как плюсы, так и минусы. С одной стороны это весьма высокая сложность разработки, но с другой производительность и эффективность решения в совокупности с хорошим заделом для будущих улучшений.

В рамках работы удолось провести лишь ограниченный анализ полученного решения. Тесты показывают концептуальную корректность и работоспособность утилиты. Однако, кроме как на непубличном языке, полноценная работоспособность не была проверенна. Это может быть исправлено в дальнейшем.

ЗАКЛЮЧЕНИЕ

Полученное решение представляет интерес, как удачный способ рассмотрения деклараций в пространстве типов. К положительным особенностям решения относятся быстрота поиска, возможность дальнейшего распараллеливания и достаточная общность решения, требующая реализации лишь одной функции для целевого языка. К минусам можно отнести возможность неучитывания некоторых специфичных особенностей системы типов целевого языка.

Полученное решение способно работать с системами типов которые имеют: номинативы, наследование, номинативы с параметризацией, механизм ограничений, функциональные типы и вариантность. Драйвер целевого языка в минимальном случае должен предоставлять декларации с типами в грамматике описанной в работе, а так же уметь отвечать на вопросы подтипизации двух типов. По хорошему не хватает анализа решения с разными языками, но это весьма объёмная работа может быть выполненна в дальнешем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Authors T. G.* Документация утиилиты dot [Электронный ресурс]. — URL: https://web.archive.org/web/20240314034809/https://graphviz.org/docs/layouts/dot/.

2 *Complete lattice*. Complete lattice — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — URL: https://web.archive.org/web/20240309222926/https://en.wikipedia.org/wiki/Complete_lattice.

3 *Denis G.* Исходный код проекта [Электронный ресурс]. — URL: https://github.com/DendyGrobovshik/bachelor-thesis.

4 *Foundation Z. S.* Официальная документация языка программирования zig [Электронный ресурс]. — URL: https://web.archive.org/web/20240511162736/https://ziglang.org/documentation/0.12.0/.

5 *JetBrains*. Дженерики и вариантность в kotlin [Электронный ресурс]. — URL: https://web.archive.org/web/20240327091959/https://kotlinlang.org/docs/generics.html.

6 *Kacper Korban A. R.* Видео презентация Inkuire [Электронный ресурс]. — URL: https://youtu.be/hPsowDgJDFo.

7 *Llimczak B.* Memory-wall problem [Электронный ресурс]. — URL: https://web.archive.org/web/20231002090943/https://developer20.com/memory-wall-problem/.

8 *Mitchell N.* Главная страница Hoogle [Электронный ресурс]. — URL: https://web.archive.org/web/20240509155056/https://hoogle.haskell.org/.

9 *Mitchell N.* Обзор Hoogle 5 [Электронный ресурс]. — URL: https://web.archive.org/web/20240203212114/https://neilmitchell.blogspot.com/2020/06/hoogle-searching-overview.html.

10 *Partially ordered set*. Partially ordered set — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — URL: https://web.archive.org/web/20240510141048/https://en.wikipedia.org/wiki/Partially_ordered_set.

11 *Radboud University Nijmegen S. T. R. G. of*. Clean wiki [Электронный ресурс]. — URL: https://web.archive.org/web/20240512102324/https://wiki.clean.cs.ru.nl/Clean.

12 *Seguin K.* Generics in zig [Электронный ресурс]. — URL: https://web.archive.org/web/20240309115114/https://www.openmymind.net/learning_zig/generics/.

13 *Staps C.* Главная страница Cloogle [Электронный ресурс]. — URL: https://web.archive.org/web/20240427163013/https://www.cloogle.org/.