Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Робототехника и комплексная автоматизация»

КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

# РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## к выпускной квалификационной работе

на тему

## «Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера»

Студент РК6-85Б _____ Никитин В.Л.
группа подпись, дата ФИО

Руководитель ВКР _____ Соколов А.П.
подпись, дата ФИО

Нормоконтролёр _____ Грошев С.В.
подпись, дата ФИО

Москва, 2024

# РЕФЕРАТ

Выпускная квалификационная работа: 49 с., 14 рис., 4 табл., 18 источн.

ТЕОРИЯ ТИПОВ, ЯЗЫКИ ПРОГРАММИРОВАНИЯ, КОМПИЛЯТОРЫ, ФУКНЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ, СИСТЕМА ТИПОВ ХИНДЛИ-МИЛНЕРА.

Работа посвящена реализации механизма вывода типов для языка программирования Kodept. Программирование выстроено вокруг глубокой математической теории. Благодаря этому появляются возможности для оптимизации, развития и улучшения языков посредством применения математики. Одним из важных применений является теория типов, которая помогает программисту в написании кода. Применение мощной системы типов позволяет зачастую снизить количество ошибок, возникающих при разработке.

**Тема работы**: *«Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера»*.

**Объект исследования**: система типов.

**Основная задача, на решение которой направлена работа**: реализация механизма вывода типов на основе выбранной системы типов в компилятора языка программирования Kodept.

**Цели работы**: развитие языка программирования Kodept посредством введения системы типов

В результате выполнения работы: 1) проведён сравнительный анализ некоторых систем типов; 2) рассмотрена система типов, которая будет применена в языке Kodept; 3) в компилятор языка Kodept добавлен механизм вывода типов на основе рассмотренной системы типов, 4) показано, что компилятор Kodept успешно выводит типы для исходного кода.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Язык программирования позволяет человеку взаимодействовать с ЭВМ. Создать язык программирования означает формализовать варианты синтаксиса, продумать семантику, спроектировать его основные возможности. Однако гораздо важнее и сложнее найти проблему, которую язык будет решать, ведь на текущий момент уже существует огромное количество языков.

В современном программировании важно уменьшать время, затраченное на разработку продукта. Сюда входит время потраченное на проектирование, разработку, отладку и поддержку продукта, а также множество других составляющих. Немаловажный вклад в оптимизацию процесса вносят выбранные инструменты: язык программирования, его экосистема, а также другие инструменты, применяемые в повседневной работе.

Нельзя не согласиться, что без компилятора не получится написать сколь-нибудь сложный проект. Он играет огромную роль, поэтому разработчикам компилятора приходится прикладывать большие усилия, чтобы язык программирования отвечал требованиям надежности и скорости. При создании инструмента такого рода важно правильно выбирать и проектировать каждую часть. Одной из основных таких частей является то, как в языке программирования взаимодействуют друг с другом данные.

В высокоуровневых языках программирования типы окружают разработчика повсюду. Чем более развитая система типов, тем больше можно выразить, используя ее, а значит, если она надежна и подкреплена математической основой, то в программе станет меньше ошибок. Кроме того, в таком случае программы можно будет применять в качестве доказательств для различных теорий [1]. Сейчас такое уже применяет компания Intel при проектировании новых алгоритмов умножения или деления.

Актуальна проблема высокого порога входа в некоторые функциональные языки программирования, например Haskell и др. Он завышен, так как в них применяются сложные математические теории, повлиявшие и на синтаксис языка, и на всю его идеологию в целом. Поэтому появилась идея создать язык программирования, вобравший в себя идеи функциональных языков, но при этом сохранивший C-подобный синтаксис. Кроме того, он задумывался еще и как способ изучить всю цепочку создания компилятора. На сегодняшний день

уже разработан синтаксис языка (Листинг 1), стабилизировано его внутреннее представление (*абстрактное синтаксическое дерево (AST)*), а также ведется работа над компилятором.

Абстрактным синтаксическим деревом называют структуру данных, получаемую после синтаксического анализа. В ней связываются элементы синтаксиса языка: параметры функции, объявление переменной и т.д.

Листинг 1 – Демонстрация синтаксиса языка Kodept

```
1   module Main =>
2   fun greeting(name: String) => "Hello " + name + "!"
3   fun main => print(greeting("world"))
```

Компилятор в языке программирования обычно разделен на несколько частей (Рис. 5). Одной из них является семантический анализ. В него входят анализ областей видимости объявлений, *проверка (и вывод) типов* и др.

**Целью** работы является совершенствование языка программирования Kodept с помощью введения системы типов с последующей реализацией механизма вывода типов в компиляторе языка Kodept.

# 1 Постановка задачи

## 1.1 Концептуальная постановка задачи

В качестве объекта исследования выбраны системы типов. Для достижения поставленной цели, определены следующие задачи:

1. провести сравнительный анализ систем типов в некоторых современных языках программирования,
2. выбрать систему типов для языка программирования Kodept,
3. описать алгоритм для вывода типов на основе выбранной системы типов,
4. расширить компилятор языка Kodept, реализовав механизм вывода типов,
5. провести тестирование для проверки правильности реализации.

# 2 Математическая постановка задачи

## 2.1 Форма Бэкуса-Наура

В работе определяются синтаксические конструкции, которые используют форму Бэкуса-Наура (англ. Backus-Naur form, BNF). Такая форма используется для записи контекстно-свободных грамматик и состоит из двух элементов: терминалов и нетерминалов. Терминалы являются примитивными элементами, в т.ч. строковым литералом, числовым и прочее. Нетерминалы могут включать в себя другие нетерминалы или терминалы.

Для определения нового нетерминала $A$ используется обозначение: $A \coloneqq B$, где $B$ может быть одной из следующих конструкций:

- конкатенация $xy$ - нетерминалы расположены последовательно (`"dog" "cat"`),
- выбор $x \mid y$ - либо $x$, либо $y$.

При этом нетерминалы в конструкции выбора будем называть вариантами. Каждый вариант может быть рассмотрен отдельно от других. Слева от знака $\coloneqq$ может быть записано несколько «примеров» определяемого нетерминала. Они могут быть использованы внутри конструкций $B$, создавая таким образом рекурсивный нетерминал.

Будем считать, что при введении нетерминалов, любой не оговоренный заранее элемент является строковым терминалом. Благодаря этому уменьшается многословность записи.

Каждый нетерминал формирует множество допустимых значений. Так, например, для нетерминала $K_1, K_2 \coloneqq * \mid K_1 \to K_2$, множеством возможных значений является $\{*, * \to *, (* \to *) \to *, \ldots\}$.

## 2.2 Теория типов

В разделе представлена информация о специальном разделе математики - теории типов [2]. Освещены важные понятия - *терм*, *тип*, *суждение* и *система типов*.

Теория типов является альтернативой для теории множеств и теории категорий. В отличие от остальных, она позволяет исследовать свойства объекта, учитывая его структуру, а не множество, которому он принадлежит. Поэто-

му теория типов нашла свое применение в программировании, в частности, в компиляторах в фазе статического анализа программы, как для вывода, так и для проверки соответствия типов. Более того, согласно изоморфизму Карри-Ховарда [3] (Табл. **??**), программы могут быть использованы для доказательства логических высказываний. Такие доказательства называют автоматическими, и они широко применяется среди таких языков, как Agda, Coq, Idris.

Терм $x$ - чаще всего элемент языка программирования, будь то переменная, константа, вызов функции и др. Термы могут включать в себя другие термы. Например, термом является конструкция $(x + 1) * (x + 1)$, построенная из других термов: $x$, 1, $+$ и $*$.

Типом $A$ обозначается метка, например объекты на натюрмортах принадлежат к типу (классу) «фрукты». Обычно каждому терму соответствует определенный тип - $x : A$. Типы позволяют строго говорить о возможных действиях над объектом, а также формализовать взаимоотношения между ними.

Система типов определяет правила взаимодействия между типами и термами. В программировании это понятие равноценно понятию типизации.

Кроме того, также используются такие термины, как *суждения* и *предположения*. С помощью суждений (англ. judgments) можно создавать логические конструкции: выражение $\vdash x : T$ говорит, что терм $x$ имеет тип $T$. Слева от знака $\vdash$ записывается контекст: $x : integer \vdash (x + 1) : integer$ - если терм $x$ имеет тип числа, то $x + 1$ тоже имеет тип числа. Таким образом, суждение обозначается так:

$$\Gamma \vdash P, \tag{2.1}$$

где $\Gamma$ - контекст,

$P$ - предположение.

Всего существует 6 видов суждений:

Тип $T$ обитаем (англ. inhabitat), если выполняется следующее: $\exists t : \Gamma \vdash t : T$ - найдется терм $t$, такой, что в контексте $\Gamma$ он будет иметь тип $T$.

Из одних суждений можно получить другие суждения по определенным правилам. Такие правила называются *правилами вывода* и выглядят следующим образом: $\frac{J_1}{J_2}$, что означает если верно суждение $J_1$, то и верно суждение $J_2$. Таким образом из правил вывода получаются деревья вывода, где каждое

| | |
|---|---|
| $\Gamma \vdash$ | $\Gamma$ - верный контекст, |
| $\Gamma \vdash \tau$ | $\tau$ - тип в контексте, |
| $\Gamma \vdash x : \tau$ | терм $x$ имеет тип $\tau$ в контексте, |
| $\vdash \Gamma = \Delta$ | контексты $\Gamma$ и $\Delta$ равны, |
| $\Gamma \vdash \tau_1 = \tau_2$ | типы $\tau_1$ и $\tau_2$ в контексте равны, |
| $\Gamma \vdash x_1 = x_2 : \tau$ | терм $x_1$ равен терму $x_2$ с типом $\tau$. |

Таблица 1 Изоморфные соотвествия между логическими высказываниями и системой типов

| **Логическое высказывание** | **Система типов** |
|---|---|
| Высказывание, $F$, $Q$ | Тип, $A$, $B$ |
| Доказательство высказывания $F$ | $x : A$ |
| Высказывание доказуемо | Тип $A$ обитаем |
| $F \implies Q$ | Функция, $A \to B$ |
| $F \wedge Q$ | Тип-произведение, $A \times B$ |
| $F \vee Q$ | Тип-сумма, $A + B$ |
| Истина | Единичный тип, $\top$ |
| Ложь | Пустой тип, $\bot$ |
| $\neg F$ | $A \to \bot$ |

входное суждение $J_1$ заменяется правилом вывода. Например, следующее правило определяет тип применения функции $f$ к аргументу $x$:

$$\frac{\Gamma \vdash x : T_1, f : T_1 \to T_2}{\Gamma \vdash f(x) : T_2} \tag{2.2}$$

Выражение 2.2 можно трактовать следующим образом: если в контексте $\Gamma$ терм $x$ имеет тип $T_1$, а терм $f$ - $T_1 \to T_2$ (функциональный тип), то можно судить, что терм $f(x)$ (применение функции) имеет тип $T_2$.

## 2.3 Классификация систем типов

Известно, что системы типов можно разделить на *динамические* и *статические* [4]. Это влияет на то, в какой момент в программе происходит проверка соответствия типов. В динамических системах - во время исполнения про-

граммы, а в статических - соответственно во время компиляции. Кроме того, существуют особые языки программирования, где все данные имеют один тип. К таким относятся многие низкоуровневые языки, например ассемблер. Все данные в нем (адреса в памяти, числа, указатели на функции) являются всего лишь последовательностью байт.

Ниже приведены основные критерии, по которым можно классифицировать систему типов в языках программирования:

1) по времени проверки соответствия типам: статическая и динамическая,
2) по поддержке неявных конверсий: сильная (англ. strong) и слабая,
3) по необходимости вручную типизировать выражение: явная и неявная.

Например, типизация в языке Python является динамической, сильной и неявной с точки зрения этой классификации [5]. Интерпретатор знает тип переменной только во время выполнения и не может неявно изменить его.

Статические системы типов обладают несомненным преимуществом, по сравнению с динамическими - компилятор может использовать накопленную во время семантического анализа информацию для оптимизации кода. Но необходимо учитывать, что такая типизация вносит некоторые неудобства: программисту постоянно приходится прилагать усилия по устранению ошибок, связанных с типами. Это делает языки со статической типизацией, хоть и более сложными в использовании, но более быстрыми, а динамически типизированным языкам приходится использовать различные специфические оптимизации вроде *JIT-компиляции*, чтобы добиться сопоставимой производительности.

JIT-компиляцией (just-in-time компиляцией) называется прием оптимизации при выполнении программы, когда компиляция происходит во время работы программы. Она была создана, чтобы решить проблемы с производительностью при интерпретации кода.

Проанализируем системы типов, используемые в некоторых современных языках программирования с целью выявить их сильные и слабые стороны.

### 2.3.1 Система типов C

C - язык программирования со статической, слабой, явной типизацией, разработанный в 1970-х годах.

Типом в языке С является интерпретация набора байт, составляющих объект [6]. Все типы бывают двух видов: базовые и производные (Рис. 1). Также их можно разделить на две группы: скалярные и агрегатные.

В группу скалярных типов относят примитивные (базовые) типы и указатели. Базовые типы в свою очередь делятся на целые и вещественные числа. Указатели - тоже скалярная величина, их размер зависит от архитектуры системы.

В группу агрегатных относятся структуры и массивы. Они позволяют определить тип, который включает в себя несколько других.

Кроме того, существуют «специальные» типы - объединения и указатели на функции. С помощью объединений можно задать варианты представления данных (Листинг 2.1).

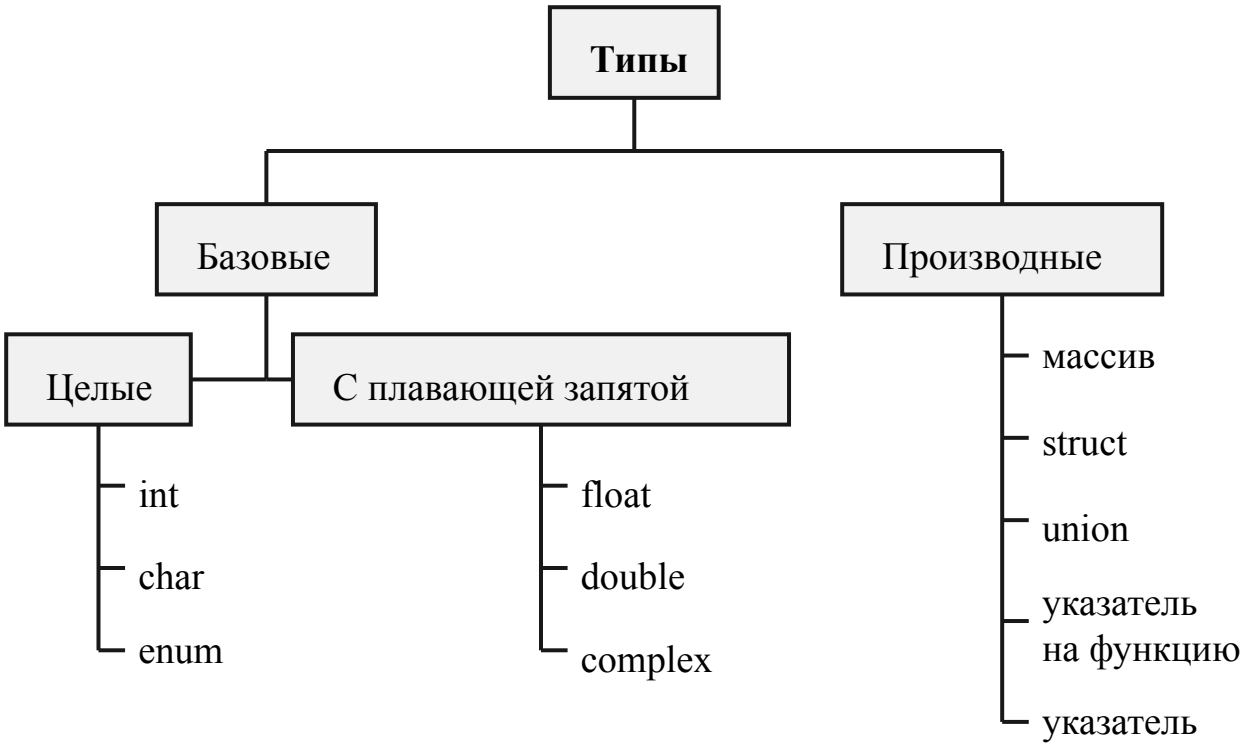


Рисунок 1 – Схематичное изображение типов в С

Листинг 2.1 – Объявление безымянного объединения в языке С. В одной переменной твкого типа может содержаться либо целое число, либо вещественное.

```
1 union {
2     int first_variant;
3     float second_variant;
4 };
```

Достоинства:

– С прост для понимания, он содержит только основные типы данных,
– язык позволяет эффективно работать с данными в том виде, как они реализованы в ЭВМ.

Недостатки:

– недостаточная выразительность по сравнению с другими языками программирования,
– мало гарантий и проверок, осуществляемых компилятором (Листинг 2.2).

Здесь под выразительностью стоит понимать то, насколько много идей можно реализовать и насколько лаконично они при этом будут выглядеть. Например, хоть в С и можно выразить идею объекто-ориентированного программирования, но это будет выглядеть гораздо более громоздко, чем в C++ [7].

Листинг 2.2 – Неправильное использование `void*` не может быть отслежено компилятором

```
1 // компилятор не знает исходный тип аргумента
2 long* foo(void* arg) { return (long*) arg; }
3
4 int main() {
5     void *value = &foo;
6     long result = (*foo(value)) + 1; // неопределенное поведение
7 }
```

### 2.3.2 Система типов Java

Язык Java разработан компанией Sun Microsystems в 1995 году. Благодаря использованию дополнительной абстракции в виде виртуальной машины, может выполняться на большом количестве архитектур ЭВМ. Популярен среди разработчиков самых разных областей: от банковского сектора до приложений под ОС Android [8].

Систему типов, применяемую в этом языке можно охарактеризовать как статическую, сильную и явную с возможностью введения неявно типизированных выражений [9]. Java создавалась под сильным влиянием идей объектно-ориентированного программирования, поэтому она включает классы, интер-

фейсы, обобщения и прочее. Язык эволюционировал, но пытался сохранить обратную совместимость с прошлыми версиями, поэтому имеются и недостатки - вся информация об обобщенной переменной стирается во время исполнения программы. Это иногда приводит к ошибкам при работе с коллекциями.

Все типы делятся на примитивные и объектные (пользовательские). Их различает значение по-умолчанию: в случае пользовательских - `null`, примитивных - в зависимости от типа. К примитивным типам относятся различные виды представления чисел, символы и логический тип. Для использования примитивных типов в коллекциях, используется упаковка (англ. boxing). Суть её заключается в том, что значение такого типа «упаковывается» в соответствующий объектный тип, который представляет собой указатель на значение вместе с метаданными.

Достоинства:

- наличие в системе типов обобщений позволяет уменьшить количество повторяемого кода,
- поддержка некоторых особенностей динамической типизации, при преобладании статической.

Недостатки:

- из-за специфики работы обобщенных типов в виртуальной машине Java, могут возникать ошибки с приведением типов,
- из-за разделения на примитивные и объектные типы, а также дополнительных затрат на упаковку, ухудшается производительность,
- избыточность определений типов в очевидных местах; хоть это и было исправлено в последующих версиях языка с помощью локального вывода типов, синтаксис языка все ещё перегружен.

### 2.3.3 Система типов ML-подобных языков

К семейству ML-подобных языков относят функциональные языки программирования, восходящие к ML (Meta Language). Этот язык был разработан Робином Милнером в 1973 году как язык для системы автоматического доказательства теорем. В основе ML лежит типизированное лямбда-исчисление - формализованная система, предложенная Алонзо Чёрчем в 1930 году [10]. Это

делает языки семейства ML статически типизированными. Кроме того, в них распространено применение алгоритмов вывода типов, поэтому они также являются неявно типизированными.

Лямбда-исчисление само по себе формально описывает некоторый набор термов, среди которых обязательно должны присутствовать 2 варианта: применение $e_1(e_2)$ и абстракция $\lambda x.e$. Функциями в лямбда-исчислении являются некие алгоритмы, в результате выполнения которых может быть получен тот или иной терм. Применение обозначает вычисление выражения $e_1$ с аргументом $e_2$ и сродни результату вызова функции. Абстракция же представляет собой способ создания новых функций посредством определения входной переменной $x$ и тела $e$. К термам лямбда-исчисления можно применять различные преобразования, в том числе $\alpha$-эквивалентность, $\beta$-редукцию и $\eta$-преобразование [11].

В отличие от обычного лямбда-исчисления, каждому терму в типизированном лямбда-исчислении сопоставляется тип. Существует множество алгоритмов для автоматической проверки типов в типизированном лямбда-исчислении. Поэтому оно используется в качестве модели для ML-подобных языков, таких как Haskell или Lisp. Кроме того, в тексте программы есть возможность отдельно не указывать типы различных конструкций: параметров функций, переменных и прочих.

Существует большое количество различных видов типизированного лямбда-исчисления, однако в 1991 году была предложена их наглядная классификация [12]. Это обобщение называется лямбда-кубом и туда входят восемь типизированных лямбда-исчислений (Рис. 2). Также на лямбда-кубе отмечены зависимости между его элементами так, что ребро обозначает усложнение системы типов путём добавления определенного свойства.

С помощью куба описываются четыре основных зависимости между типами и термами (в скобках приведены обозначения соответствующих систем типов, где такое свойство выполняется):

- термы зависят от термов ($\lambda^{\rightarrow}$),
- термы зависят от типов ($\lambda 2$),
- типы завися от типов ($\lambda\underline{\omega}$),

– типы зависят от термов ($\lambda P$).

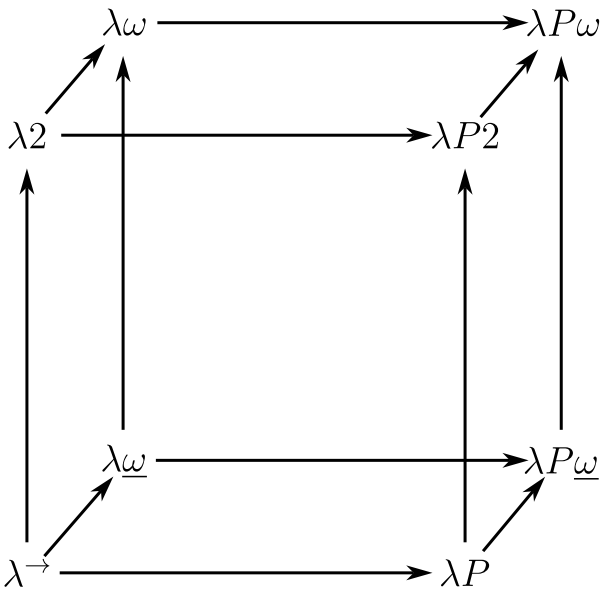


Рисунок 2 – Графическое изображение лямбда-куба

Согласно рисунку 2, система $\lambda^{\rightarrow}$ (просто типизованное лямбда-исчисление) является самой простой и составляет базу для всех остальных систем. Рассмотрим её подробнее. В множество возможных типов входит переменная типа $\alpha$ и функциональный тип $\tau_1 \rightarrow \tau_2$. Таким образом, $\tau := \alpha \mid \tau_1 \rightarrow \tau_2$. В множество термов входят переменная $x$, применение $e_1(e_2)$ и абстракция $\lambda x : \tau.e$. Такая система типов во многом похожа на систему типов в языке программирования C, за исключением пользовательских типов, массивов и указателей.

В систему $\lambda 2$ (полиморфное лямбда-исчисление), по сравнению с предыдущей системой, добавляется так называемый полиморфный тип $\sigma := \forall\alpha.\tau \mid \tau$. Таким образом, терм может зависеть от конкретного типа $\tau$: $f : \forall\alpha.\alpha \rightarrow \alpha \implies f(\tau) : \tau \rightarrow \tau$. Абстракции с полиморфным типом похожи на шаблонные функции из языка C++.

Зависимость типов от типов можно понимать как функции (операторы) над типами. Простейшим примером будет функция из типа $\alpha$ в тип $\alpha$: $f : \alpha \rightarrow \alpha$. Более формально записывают так: $f \equiv \lambda\alpha : *.\alpha \rightarrow \alpha$, где $*$ обозначает категорию типов. Однако возникает проблема, что функция $f$ не является ни типом,

ни термом. Решением этой проблемы является добавление новой категории $K$ видов (англ. kinds):

$$K \coloneqq * \mid * \to *, \tag{2.3}$$

где $*$ - категория типов.

Функции, похожие на рассмотренную функцию $f$, называют конструкторами типа. С точки зрения языка C++, такие функции можно сравнить с шаблонными классами (структурами). Действительно, шаблонный класс определяет шаблон типа (срав. вида), конкретный экземпляр которого является типом.

Зависимость типов от термов является гораздо более сложной, по сравнению с уже рассмотренными и не имеет аналогов в привычных языках программирования. Положим функцию $f : \tau \to *$, где $*$ - ранее рассмотренная категория типов. Тогда $f$ является конструктором, а $(\lambda a : \tau.f(a)) : *$, где $a$ - терм с типом $\tau$. Таким образом функция $f$ продуцирует зависимые от термов типы. Такой подход может быть использован для явного определения контрактов функции в программировании. Например, функция деления, оба аргумента которой являются числом, но второй отличен от нуля. Такой контракт выносится на уровень сигнатуры функции, что позволяет строго ему следовать.

Остальные вершины куба формируются комбинацией уже рассмотренных систем типов в порядке, формируемом направлениями рёбер. Например, система $\lambda P\omega$ является наиболее полной и в ней выполняются все четыре зависимости. Кроме того, это можно показать с помощью таблицы 2.

Таблица 2 Наличие зависимостей между категорией типов ($*$) и категорией видов ($\Box$) в системах типов, описанных в лямбда-кубе.

| **Система типов** | **Используемые правила** |
|---|---|
| $\lambda^{\to}$ | $(*, *)$ |
| $\lambda 2$ | $(*, *)$, $(\Box, *)$ |
| $\lambda P$ | $(*, *)$, $(*, \Box)$ |
| $\lambda\underline{\omega}$ | $(*, *)$, $(\Box, \Box)$ |
| $\lambda\omega$ | $(*, *)$, $(\Box, *)$, $(\Box, \Box)$ |
| $\lambda P\underline{\omega}$ | $(*, *)$, $(*, \Box)$, $(\Box, \Box)$ |
| $\lambda P\omega$ | $(*, *)$, $(\Box, *)$, $(*, \Box)$, $(\Box, \Box)$ |

Конструкция вида $(s_1, s_2)$, где $s \in \{*, \square\}$ означает следующее:

$$\frac{\Gamma \vdash A : s_1 \;\; \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\prod x : A.B) : s_2}, \tag{2.4}$$

где $\prod x : A.B$ означает декартово произведение всех типов (видов) $B$, образованных от переменной $x$ с типом (видом) $A$.

Достоинства:

- разработчик может выразить больше инвариантов, контрактов и др. в сигнатуре функции,
- математически строгое обоснование корректности и надёжности,
- компилятор, использующий эту систему типов, имеет больше возможностей по обнаружению ошибок.

Недостатки:

- использование дополнительного синтаксиса при обозначении типов может ухудшить читаемость, а также усложнить написание кода,
- может значительно увеличить время компиляции из-за обилия проверок,
- неявная типизация может тоже повлечь ухудшение читаемости, так как программисту необходимо выводить типы самому; однако это исправляется использованием сред разработки.

В итоге была проведена классификация некоторых систем типов с описанием их достоинств и недостатков. Особое внимание далее уделяется системе типов Хиндли-Милнера, являющейся применением $\lambda 2$ в языках программирования. Различные ее модификации широко используются в языках ML-группы за счёт того, что используя эту систему типов, можно автоматически выводить тип термов, одновременно обеспечивая статическую типизацию. В качестве системы типов для языка Kodept приводится модификация системы типов Хиндли-Милнера.

## 2.4 Система типов Хиндли-Милнера

Для начала определим термы:

$$e_1, e_2, e_3 \coloneqq x \tag{2.5a}$$

$$| \; e_1(e_2) \tag{2.5b}$$

$$| \; \lambda x : \tau . e_1 \tag{2.5c}$$

$$| \; \text{let } x : \tau = e_2 \text{ in } e_2 \tag{2.5d}$$

$$| \; \text{:num:} \tag{2.5e}$$

$$| \; (e_1, e_2) \tag{2.5f}$$

$$| \; \text{if } e_1 \text{ then } e_2 \text{ otherwise } e_3, \tag{2.5g}$$

где $e_1, e_2, e_3$ - термы,

$x$ - имя переменной (англ. binding),

:num: - числовой литерал,

$\tau$ - тип.

Поясним значение некоторых вариантов терма подробнее:

(2.5a) переменная позволяет сослаться на другое имя, определенное выше по тексту программы, например на функцию или другую переменную.

(2.5b) запись $e_1(e_2)$ обозначает применение функции ($e_1$) к ее аргументу ($e_2$); проще говоря - вызов функции.

(2.5c) $\lambda x : \tau . e_1$ означает создание новой лямбда-функции с аргументом $x$ и телом $e_1$.

(2.5d) объявление новых переменных происходит с помощью конструкции let ... in ....

(2.5f) конструкция необходима для создания пар из двух других термов, объединяя их в кортеж,

(2.5g) этот вариант необходим для условного выполнения термов, при этом типы термов $e_2$ и $e_3$ должны оказаться одинаковыми.

В некоторых элементах к переменной $x$ приписывается ограничение на тип $\tau$ следующим образом $x : \tau$. Это необходимо для того, чтобы явно указать требуемый тип, как это делается в языках программирования с явной типизацией, например C (Листинг 2.3).

Листинг 2.3 – Явное указание типа аргумента в языке C.

```
1 int foo(int a) {
2     return a;
3 }
```

Далее введено определение типа $\tau$.

$$\begin{aligned} \tau_1, \tau_2 \coloneqq & \text{ Integer} \mid \text{Real} \mid \text{Boolean} \\ & \mid \alpha \\ & \mid \tau_1 \to \tau_2 \\ & \mid (\tau_1, \tau_2, \ldots, \tau_n) \\ & \mid C, \end{aligned} \tag{2.6}$$

где $C$ - имя типа (константа), определенное пользователем,

$\alpha$ - переменная типа,

Integer, Real, Boolean - примитивные типы для целых, вещественных и логических данных соответственно.

Переменная типа необходима для тех же целей, что и обычная переменная: она может ссылаться на другой тип или быть любым типом. Под записью $(\tau_1, \tau_2, \ldots, \tau_n)$ стоит понимать тип-кортеж, образованный несколькими другими типами.

Также к обычным типам необходимо добавить так называемые *полиморфные* типы (2.7). Они необходимы для введения квантора всеобщности по отношению к переменным типа. С точки зрения обычных языков программирования, такие полиморфные типы можно оценивать как обобщенные типы (Листинг 2.4).

$$\sigma \coloneqq \tau \mid \forall A.\sigma, \tag{2.7}$$

где $A = \{\alpha\}$ - неупорядоченное множество переменных типа.

Листинг 2.4 – Определение обобщенной функции в C++

```
1  template<typename T>
2  void foo(T value) {}
```

Далее введем следующие понятия: *контекст* $\Gamma$ (2.8), множество *свободных типов* (англ. free types) (2.9), *обобщение* (англ. generalize) (2.10), *подстановка* (англ. substitutions) (2.11) и *конкретизация* (англ. instantiate) (2.13). В скобках указан номер выражения, содержащего необходимое определение.

$$\Gamma = \{x : \sigma \mid x \in X, \sigma \in \Sigma\}, \tag{2.8}$$

где $X$ - множество термов,

$\Sigma$ - множество полиморфных типов.

$$\begin{aligned} ft(\sigma) &= ft(\tau) \backslash A, \\ ft(\tau) &= \{\alpha \mid \alpha \in \tau\}, \\ ft(\Gamma) &= \bigcup_{x:\sigma \in \Gamma} ft(\sigma) \end{aligned} \tag{2.9}$$

где $\alpha \in \tau$ - переменная типа, использованная в типе $\tau$,

$X \backslash Y$ - множество $X$, исключая элементы множества $Y$.

$$gn(\Gamma, \tau) = \forall A.\tau, \tag{2.10}$$

где $A = ft(\tau) \backslash ft(\Gamma)$.

$$\mathcal{S} = [\alpha_1 \coloneqq \tau_1, \alpha_2 \coloneqq \tau_2, \ldots \alpha_n \coloneqq \tau_n], \tag{2.11}$$

где ни одна пара $\alpha_i \coloneqq \tau_i$ не должна быть вида $\alpha_i \coloneqq \alpha_i$, иначе будет получена подстановка для бесконечно рекурсивного типа.

Композиция подстановок:

$$\mathcal{S}_1 \circ \mathcal{S}_2 = \mathcal{S}_1 \cup [\alpha_i \coloneqq \mathcal{S}_1 \tau_i] \tag{2.12}$$

Применение подстановки $\mathcal{S}\tau$ - операция замены всех вхождений очередной $\alpha_i$ из подстановки $\mathcal{S}$ на $\tau_i$ в типе $\tau$.

Будем называть $\beta$ - уникальную переменную типа.

$$it(\sigma) = \mathcal{S}_\sigma \tau, \tag{2.13}$$

где $\mathcal{S}_\sigma = [\alpha \coloneqq \beta \mid \alpha \in A, \beta \neq \alpha]$.

Правила вывода, разработанные Милнером [13], позволяют получить доказательство, что любой терм, заданный выражением (2.5) можно типизировать определенным типом. Алгоритм, построенный с такими правилами, называется алгоритмом $\mathcal{W}$. Одним из его недостатков является неточное определение места ошибки при типизации выражения. Вместо него предлагается использовать модификацию этого алгоритма с использованием ограничений (англ. constraints) и отложенной унификацией.

Унификацией называется процесс поиска такой подстановки $\mathcal{S}$ для двух типов $\tau_1$ и $\tau_2$, что $\mathcal{S}\tau_1 = \mathcal{S}\tau_2$. Для решения этой задачи существует алгоритм $\mathcal{U}$:

$$\begin{aligned}
\mathcal{U}(\tau_1, \tau_1) &= []\,, \\
\mathcal{U}(\alpha_1, \tau_2) &= [\alpha_1 \coloneqq \tau_2] \text{ если } \alpha_1 \notin \tau_2, \\
\mathcal{U}(\tau_1, \alpha_2) &= [\alpha_2 \coloneqq \tau_1] \text{ если } \alpha_2 \notin \tau_1, \\
\mathcal{U}(\tau_1^{in} \to \tau_1^{out}, \tau_2^{in} \to \tau_2^{out}) &= \mathcal{U}(\tau_1^{in}, \tau_2^{in}) \cup \mathcal{U}(\tau_1^{out}, \tau_2^{out}), \\
\mathcal{U}((\tau_1^A, \ldots, \tau_n^A), (\tau_1^B, \ldots, \tau_n^B)) &= \bigcup_{i=1}^{n} \mathcal{U}(\tau_i^A, \tau_i^B)
\end{aligned} \tag{2.14}$$

## 2.5 Использование ограничений при выводе типов

Алгоритм вывода типов, использующий ограничения - алгоритм $\mathcal{W}_c$ - имеет, по сравнению с алгоритмом $\mathcal{W}$, два основных преимущества:

– он использует отложенную унификацию и вместо нее работает с ограничениями, а не с подстановками, что позволяет выводить тип для более узких случаев,

– ему не нужен глобальный контекст при выводе типа - всю требуемую информацию он сохраняет в ограничениях и *множестве предположений*.

Под ограничением $c$ будем понимать следующее определение:

$$\begin{aligned}
c \coloneqq\ & \tau_1 \equiv \tau_2, \\
\mid\ & \tau_1 \leqslant_{\mathcal{M}} \tau_2, \\
\mid\ & \tau \preceq \sigma
\end{aligned} \tag{2.15}$$

– Ограничение эквивалентности ($\tau_1 \equiv \tau_2$) говорит, что типы $\tau_1$ и $\tau_2$ должны быть унифицированы.

– Явное ограничение на экземпляр ($\tau \preceq \sigma$) указывает, что тип $\tau$ должен быть каким-то конкретным экземпляром полиморфного типа $\sigma$.

– Неявное ограничение на экземпляр ($\tau_1 \leqslant_{\mathcal{M}} \tau_2$) показывает, что тип $\tau$ должен быть конкретным экземпляром типа, полученного после обобщения типа $\tau_2$ в контексте $\mathcal{M}$.

Последние два ограничения возникают из-за полиморфных свойств объявления переменной. А именно - тип переменной $x$ в выражении let $x = e_1$ in $e_2$

должен быть конкретным экземпляром полиморфного типа для каждого места использования. Явное ограничение на экземпляр не подходит, так как на момент обработки выражения тип $e_1$ не может быть полиморфным. Поэтому необходимо использовать неявное ограничение, контекст $\mathcal{M}$ которого пополняется по ходу выполнения алгоритма.

Правила вывода, используемые в алгоритме $\mathcal{W}_c$, состоят из суждений вида $\mathcal{A}, \mathcal{C} \vdash e : \tau$, где $\mathcal{A}$ - множество предположений, $\mathcal{C}$ - ограничения, $e$ - терм, $\tau$ - тип. Сами правила представлены в подразделе 2.5.1 В отличие от контекста $\Gamma$, используемом в алгоритме $\mathcal{W}$, в множество предположений для каждой переменной сохраняется набор её возможных типов (2.16). Кроме того, при рекурсивном применении правил, множества предположений просто соединяются между собой.

$$\mathcal{A} = \{x : T \mid x \in X\}, \tag{2.16}$$

где $X$ - множество переменных,

$T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ - множество возможных типов.

Неявное ограничение на экземпляр использует контекст $\mathcal{M}$. Он может быть получен для каждого терма $e$ следующим образом:

$$\begin{aligned}
\mathcal{M}(x) &= \emptyset, \\
\mathcal{M}(e_1(e_2)) &= \mathcal{M}(e_1) \cup \mathcal{M}(e_2), \\
\mathcal{M}(\lambda x : \tau . e_1) &= \{\beta\} \cup \mathcal{M}(e_1), \\
\mathcal{M}(\text{let } x : \tau = e_1 \text{ in } e_2) &= \mathcal{M}(e_1) \cup \mathcal{M}(e_2), \\
\mathcal{M}(\text{:num:}) &= \emptyset, \\
\mathcal{M}((e_1, e_2)) &= \mathcal{M}(e_1) \cup \mathcal{M}(e_2), \\
\mathcal{M}(\text{if } e_1 \text{ then } e_2 \text{ otherwise } e_3) &= \mathcal{M}(e_1) \cup \mathcal{M}(e_2) \cup \mathcal{M}(e_3),
\end{aligned} \tag{2.17}$$

### 2.5.1 Правила вывода

Для того, чтобы для любого терма определить его тип, необходимо задать правила вывода. Ниже представлены такие правила для каждого варианта терма (2.5). Правила вывода приведены в том виде, который определен в разделе 2.2.

Правило для варианта терма $e = x$:

$$\frac{}{\{x : \{\beta\}\}, \emptyset \vdash e : \beta} \tag{2.18}$$

Правило для варианта терма $e = e_1(e_2)$:

$$\frac{\mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \;\; \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\} \vdash e : \beta} \tag{2.19}$$

Правило для варианта терма $e = \lambda x : \tau . e_1$:

$$\frac{\mathcal{A}, \mathcal{C} \vdash e_1 : \tau''}{\mathcal{A} \backslash x, \mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\} \cup \{\beta \equiv \tau\} \vdash e : (\beta \to \tau'')} \tag{2.20}$$

Правило для варианта терма $e =$ let $x : \tau = e_1$ in $e_2$:

$$\frac{\mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \;\; \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2 \backslash x, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leqslant_{\mathcal{M}} \tau_1 \mid x : \tau' \in \mathcal{A}_2\} \cup \{\tau \leqslant_{\mathcal{M}} \tau_1\} \vdash e : \tau_2} \tag{2.21}$$

Правило для варианта терма $e =$ :num::

$$\frac{}{\emptyset, \emptyset, \vdash e : \text{Integer}} \tag{2.22}$$

Правило для варианта терма $e = (e_1, e_2)$:

$$\frac{\mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \;\; \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \vdash e : (\tau_1, \tau_2)} \tag{2.23}$$

Правило для варианта терма $e =$ if $e_1$ then $e_2$ otherwise $e_3$:

$$\frac{\mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \;\; \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \;\; \mathcal{A}_3, \mathcal{C}_3 \vdash e_3 : \tau_3}{\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \equiv \text{Boolean}, \tau_2 \equiv \tau_3\} \vdash e : \tau_3} \tag{2.24}$$

Продемонстрируем применение правил вывода на примере следующего выражения:

$$\begin{aligned} &\lambda w. \text{ let } y = w \\ &\quad \text{in let } x = y(0) \\ &\quad \text{in } x \end{aligned} \tag{2.25}$$

В этом выражении присутствует лямбда-функция, поэтому все места использования переменной $w$ должны иметь одинаковый тип. В результате применения правил (), получим набор ограничений 2.26 и предварительный тип терма - $\tau_5 \to \tau_4$. В нем содержится два неявных ограничения, образованных использованием конструкции let ..., с контекстом $\mathcal{M} = \{\tau_5\}$ - типом переменной $w$.

$$\begin{aligned} \mathcal{C}_{example} = &\{\tau_2 \equiv \text{Integer} \to \tau_3\} \\ &\cup \left\{\tau_4 \leqslant_{\{\tau_5\}} \tau_3\right\} \\ &\cup \left\{\tau_2 \leqslant_{\{\tau_5\}} \tau_1\right\} \\ &\cup \{\tau_5 \equiv \tau_1\} \end{aligned} \tag{2.26}$$

## 2.6 Решение ограничений

После применения правил вывода к терму, получаются множества ограничений и предположений. В то время как множество предположений не требует дополнительной обработки, множество ограничений необходимо разрешить, используя специальный алгоритм. В результате будет получена такая подстановка $\mathcal{S}$, которая будет верно типизировать заданный терм.

К набору ограничений сама по себе может быть применена подстановка согласно следующему правилу:

$$\begin{aligned} \mathcal{S}(\tau_1 \equiv \tau_2) &= \mathcal{S}\tau_1 \equiv \mathcal{S}\tau_2, \\ \mathcal{S}(\tau \preceq \sigma) &= \mathcal{S}\tau \preceq \mathcal{S}\sigma, \\ \mathcal{S}(\tau_1 \leqslant_{\mathcal{M}} \tau_2) &= \mathcal{S}\tau_1 \leqslant_{\mathcal{SM}} \mathcal{S}\tau_2, \end{aligned} \tag{2.27}$$

где $\mathcal{SM}$ - применение подстановки к каждому типу из множества $\mathcal{M}$,
$\mathcal{S}\sigma = \forall A.\mathcal{S}\tau$.

Определим, какие переменные типа активны в текущем контексте используя 2.9:

$$\begin{aligned} active(\tau_1 \equiv \tau_2) &= ft(\tau_1) \cup ft(\tau_2), \\ active(\tau \preceq \sigma) &= ft(\tau) \cup ft(\sigma), \\ active(\tau_1 \leqslant_{\mathcal{M}} \tau_2) &= ft(\tau_1) \cup (ft(\mathcal{M}) \cap ft(\tau_2)) \end{aligned} \tag{2.28}$$

Наконец, определим алгоритм решения множества ограничений (2.29). На вход он принимает набор ограничений, а в результате будет получена подстановка. Из алгоритма видно, что явное и неявное ограничения сводятся к форме ограничения эквивалентности, после чего из него получается подстановка.

$$
\begin{aligned}
solve(\emptyset) &= \emptyset, \\
solve(\{\tau_1 \equiv \tau_2\} \cup \mathcal{C}) &= solve(\mathcal{SC}) \circ \mathcal{S}, \\
solve(\{\tau \preceq \sigma\} \cup \mathcal{C}) &= solve(\{\tau \equiv it(\sigma)\} \cup C), \\
solve(\{\tau_1 \leqslant_{\mathcal{M}} \tau_2\} \cup \mathcal{C}) &= solve(\{\tau_1 \preceq gn(\mathcal{M}, \tau_2)\} \cup C) \\
&\quad \text{если } (ft(\tau_2) \backslash \mathcal{M}) \cap active(\mathcal{C}) = \emptyset,
\end{aligned}
\tag{2.29}
$$

где $\mathcal{S} = \mathcal{U}(\tau_1, \tau_2)$.

Рассмотрим решение множества ограничений 2.26:

$$
\begin{aligned}
&solve(\mathcal{C}_{example}) = \\
&= solve(\left\{\tau_2 \equiv \mathrm{I} \to \tau_3, \tau_4 \leqslant_{\{\tau_5\}} \tau_3, \tau_2 \leqslant_{\{\tau_5\}} \tau_1, \tau_5 \equiv \tau_1\right\}) && = \\
&= solve(\left\{\tau_4 \leqslant_{\{\tau_5\}} \tau_3, \mathrm{I} \to \tau_3 \leqslant_{\{\tau_5\}} \tau_1, \tau_5 \equiv \tau_1\right\}) \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\left\{\tau_4 \leqslant_{\{\tau_1\}} \tau_3, \mathrm{I} \to \tau_3 \leqslant_{\{\tau_1\}} \tau_1\right\}) \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\left\{\tau_4 \leqslant_{\{\tau_1\}} \tau_3, \mathrm{I} \to \tau_3 \preceq \tau_1\right\}) \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\left\{\tau_4 \leqslant_{\{\tau_1\}} \tau_3, \mathrm{I} \to \tau_3 \equiv \tau_1\right\}) \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\left\{\tau_4 \leqslant_{\{\tau_3\}} \tau_3\right\}) \circ [\tau_1 \coloneqq \mathrm{I} \to \tau_3] \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\{\tau_4 \preceq \tau_3\}) \circ [\tau_1 \coloneqq \mathrm{I} \to \tau_3] \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\{\tau_4 \equiv \tau_3\}) \circ [\tau_1 \coloneqq \mathrm{I} \to \tau_3] \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= solve(\emptyset) \circ [\tau_4 \coloneqq \tau_3] \circ [\tau_1 \coloneqq \mathrm{I} \to \tau_3] \circ [\tau_5 \coloneqq \tau_1] \circ [\tau_2 \coloneqq \mathrm{I} \to \tau_3] && = \\
&= [\tau_4 \coloneqq \tau_3, \tau_1 \coloneqq \mathrm{I} \to \tau_3, \tau_5 \coloneqq \mathrm{I} \to \tau_3, \tau_2 \coloneqq \mathrm{I} \to \tau_3],
\end{aligned}
\tag{2.30}
$$

где $\mathrm{I} = \mathrm{Integer}$.

Применив полученную подстановку к предварительному типу $\tau_5 \to \tau_4$, получим наиболее общий тип выражения 2.25: $(\mathrm{Integer} \to \tau_3) \to \tau_3$. Как альтернативу полученному типу, можно привести следующий пример из языка C++:

Листинг 2.5 – Вид полученного типа с точки зрения языка C++.

```
1    template<typename T>
2    using ResultingType = T (std::function<T (int)>)(*);
```

Согласно определению 2.29, существует возможность, что два неявных ограничения на экземпляр будут зависимы друг от друга. Например, $\mathcal{C} = \{\tau_1 \leqslant_\emptyset \tau_2, \tau_2 \leqslant_\emptyset \tau_1\}$. В таком случае обе переменных типа являются активными (согласно 2.28) и множество ограничений не может быть разрешено. Однако такое множество не может быть создано путём работы алгоритма $\mathcal{W}_c$.

## 2.7 Алгоритм вывода типов $\mathcal{W}_c$

Используя рассмотренные выше определения и алгоритмы, можно определить и сам алгоритм для вывода типов $\mathcal{W}_c$. Как уже было сказано, он использует правила вывода из раздела 2.5.1 и алгоритм решения ограничений. На вход он принимает контекст $\Gamma$ и терм $e$, возвращает - подстановку $\mathcal{S}$ и тип терма $\tau$.

Введём явное ограничение на экземпляр и для множеств $\mathcal{A}$ и $\Gamma$ (2.31). Благодаря этому определению, типизируемый терм может использовать переменные, определенные вне него. Например, пусть $\mathcal{A} = \{id : \tau_2, id : \tau_3, f : \tau_4\}$, а $\Gamma = \{id : \forall\{\alpha_1\}.\alpha_1 \to \alpha_1, f : \tau_1 \to \tau_1\}$. Тогда $\mathcal{A} \preceq \Gamma = \{\tau_2 \preceq \forall\{\alpha_1\}.\alpha_1 \to \alpha_1, \tau_3 \preceq \forall\{\alpha_1\}.\alpha_1 \to \alpha_1, \tau_4 \preceq \tau_1 \to \tau_1\}$. Таким образом внешние переменные правильно используются в алгоритме.

$$\mathcal{A} \preceq \Gamma = \{\tau \preceq \sigma \mid x : \tau \in \mathcal{A}, x : \sigma \in \Gamma\} \tag{2.31}$$

На рисунке 3 представлена блок-схема рассматриваемого алгоритма. Из нее видно, что процесс применения правил вывода не зависит от использования контекста $\Gamma$, что позволяет типизировать любой терм, вне зависимости от внешних переменных. Для определения того, что все переменные из множества предположений определены, используется условие $dom(\mathcal{A}) \nsubseteq dom(\Gamma) = \{x \mid x : \tau \in \mathcal{A}\} \nsubseteq \{x \mid x : \sigma \in \Gamma\}$.

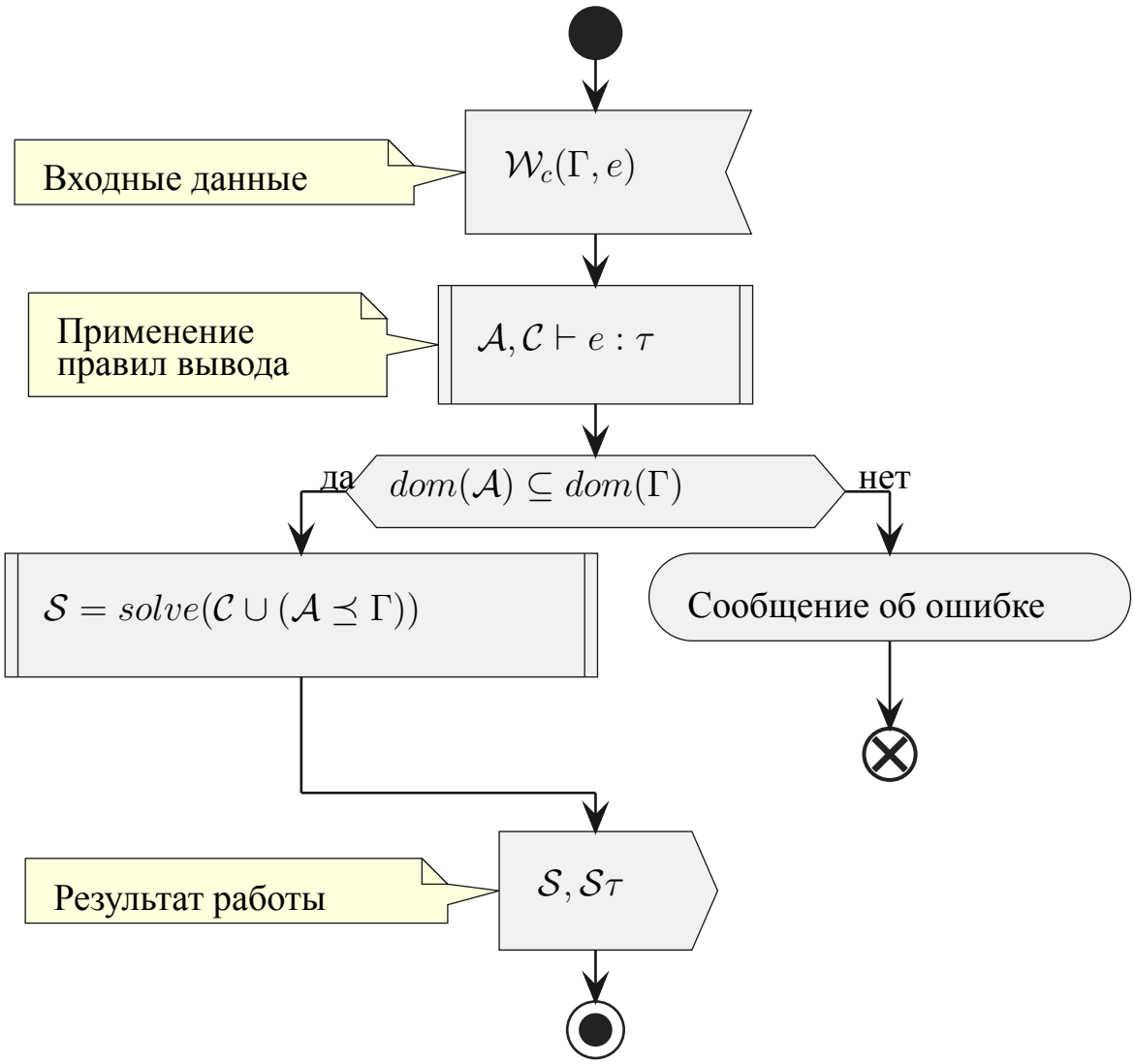


Рисунок 3 – Блок-схема алгоритма $\mathcal{W}_c$

# 3 Программная реализация

## 3.1 Варианты использования

Проект Kodept разрабатывается с использованием языка программирования Rust. Этот язык предлагает надежный концепт управления памятью, не имея при этом сборщика мусора [14]. Кроме того, он соперничает по скорости с C и C++ и применяется в довольно широком спектре приложений. Основные преимущества выбора этого языка:

- Rust работает быстрее за счёт использования мощных оптимизаторов, а так же применяет более строгие требования к разработке в целом,
- он предоставляет больше гарантий разработчику, как посредством его системы типов, так и другими средствами, например, borrow checker [15],
- система сборки создает нативный файл программы - его можно запустить, не имея на машине специальных сред выполнения.

Были составлены варианты использования компилятора с точки зрения как пользователя программы, так и разработчика компилятора (Рис. 4). Разработчик компилятора выделен как отдельное действующее лицо, так как ему необходим доступ к расширенной информации о работе процесса компиляции. Важно понимать, что компилятор всё ещё находится в разработке, поэтому в будущем будет добавлено больше вариантов использования. Реализация существующих вариантов выполнена в виде набора команд и флагов для интерфейса командной строки.

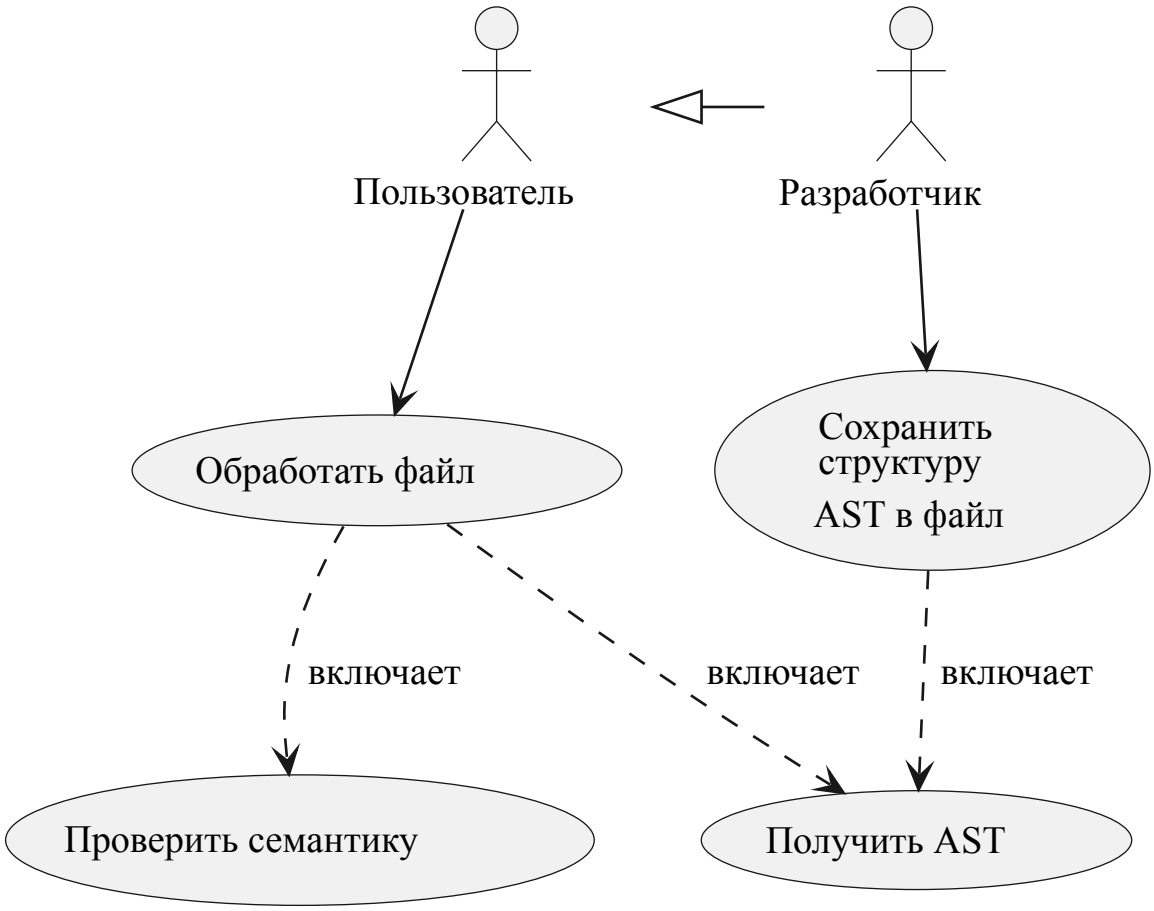


Рисунок 4 – UML-диаграмма вариантов использования компилятора языка Kodept

Рассмотрим варианты использования подробнее. Пользователь программы взаимодействует с ней посредством интерфейса командной строки (англ. command line interface, CLI). При этом на выбор ему доступны две команды, отражающие соответсвующий вариант использования: для обработки файла - команда `run`, для сохранения структуры AST - команда `graph`. Приведём пример использования CLI:

```
kodept graph -d -o output examples/church.kd
```

имя исполняемого файла — команда — опции запуска — путь до исходного файла

В результате работы команды `graph` будет получен файл, содержащий представление AST на языке `dot` [16]. Примером отрисовки такого файла является рисунок А.1. В процессе выполнения команды компилятор выполняет первые два шага - синтаксический и лексический анализы, а затем строит абстрактное синтаксическое дерево. Эта команда подразумевается в отладочных целях.

Другая команда - `run` является более комплексной, по сравнению с предыдущей. В результате её выполнения будут выведены все собранные *диагностики*. Во время работы команда выполняет лексический, синтаксический и семантический анализы, в том числе и вывод типов. В будущем подразумевается, что она будет проводить полный цикл компиляции (Рис. 5). Кроме команд, используются опции компиляции, полный список которых приведён в таблице 3.

Таблица 3 Опции командной строки, используемые в компиляторе языка Kodept

| **Короткое название** | **Полное название** | **Описание** |
|---|---|---|
| `-d` | `--debug` | Включение отладочной печати |
| `-v` | `--verbose` | Включение подробной печати |
| `-s` | `--severity` | Установка уровня логирования по-умолчанию |
| `-o` | `--out` | Путь для записи выходных данных |
| | `--style` | Стиль отображения диагностик |
| | `--tab-width` | Добавление отступов |
| `-c` | `--color` | Использование цветной печати |
| | `--disable-diagnostics` | Отключить вывод диагностик |
| | `--stdin` | Чтение входных данных из потока стандартного ввода |
| `-e` | `--extension` | Использовать указанное расширение для файлов |
| `-V` | `--version` | Получить версию программы |

Диагностика (диагностическое сообщение) - сообщение компилятора, выводимое в стандартный поток ошибок. Она необходима для указания проблемы, ошибки или другой вспомогательной информации об исходном коде. Диагностики формируются в нескольких местах при выполнении программы. Например, после синтаксического анализа, все встреченные ошибки синтакси-

са трансформируются в диагностики. Ниже приведен пример диагностики для некоторой функции `test` на языке Kodept.

```
    note[TC001]: `test` inferred to: Floating -> Floating
  ┌─ examples/test.kd:5:9
5 │     fun test(m) {
  │         ^^^^
```

Формат диагностики может меняться в зависимости от опций компиляции, но упрощённо выглядит следующим образом:

```
bug[TC002]: Unimplemented
```
важность — код — основное описание

```
examples/test.kd:9:10
```
место в исходном файле

```
// code...

    TTTT

here
```
дополнительное описание

где важность - уровень важности диагностики, может быть `note`, `info`, `error` и пр.,
код - пятибуквенный номер, идентифицирующий диагностику.

## 3.2 Архитектура компилятора

В структуре программы почти любого компилятора можно выделить 3 основные части [17]. Следует отметить, что, хоть названия частей и похожи, они не имеют мало общего с одноимёнными названиями из сферы веб-разработки. На рисунке 5 также отмечены внутренние элементы каждой из частей, а стрелки показывают данные, передаваемые от стадии к стадии. Результатом компиляции является объектный модуль (файл), в котором содержатся набор символов, отладочная информация, константные данные и др. Символы в объектном файле представляют собой именованные объекты, позволяющие идентифицировать функции, переменные, типы данных.

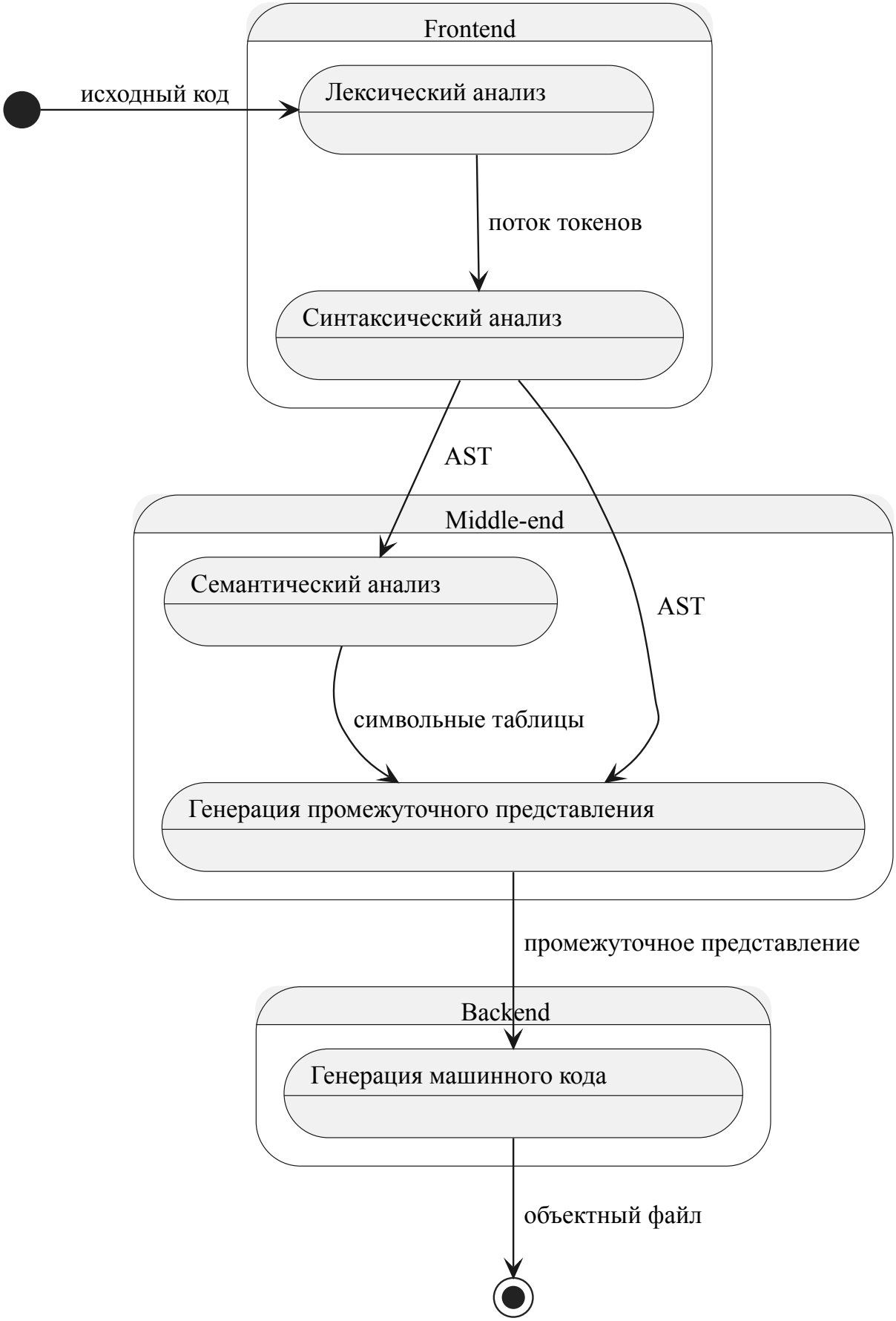


Рисунок 5 – Схема состояний работы компилятора

Реализация системы типов, описанной в разделе 2.4, в языке Kodept является одной из основных задач этой работы. Поэтому будет подробно рассмотрен семантический анализатор, включающий механизм вывода типов. Процесс разработки frontend части компилятора был завершён до этой работы и подробно раскрываться не будет.

На рисунке 6 дано более подробное описание семантического анализатора. При этом сплошная стрелка отображает ассоциацию между объектами, а штриховая - зависимость в направлении от зависимого к главному. С помощью записи `0..*` у объекта указывается, что таких объектов может быть от 0 и больше.

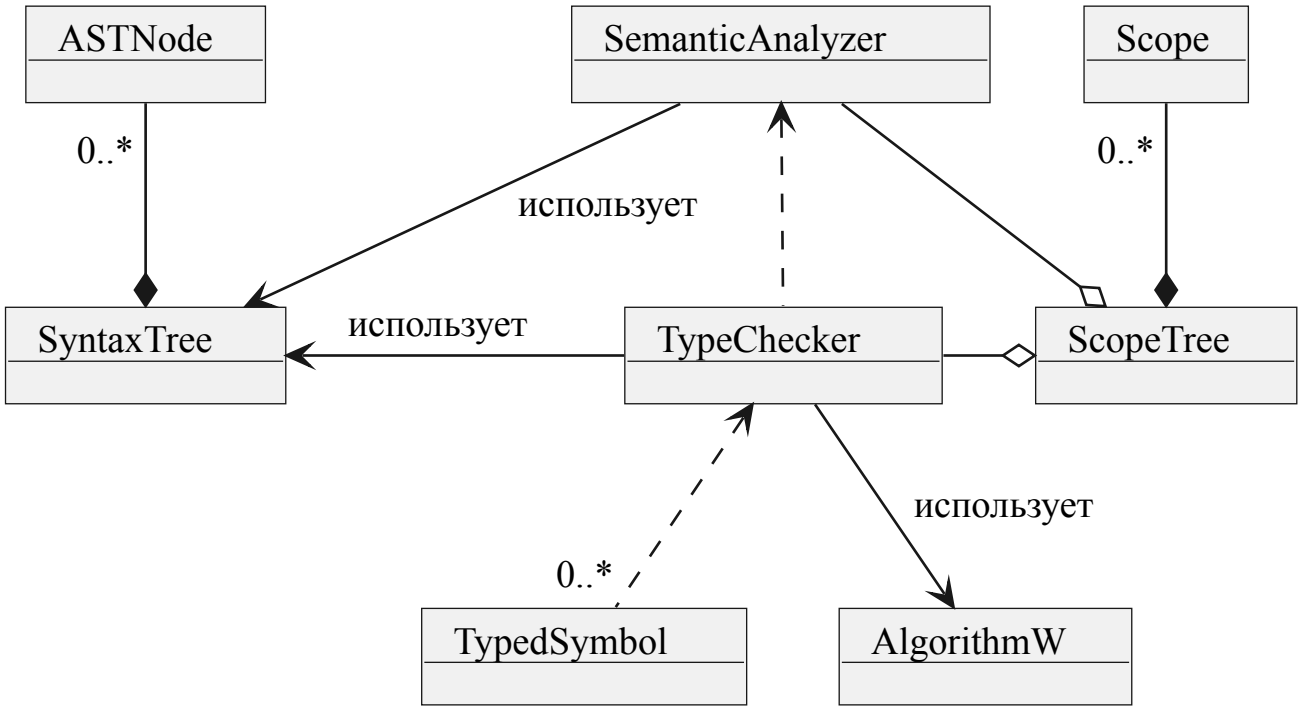


Рисунок 6 – UML-диаграмма классов, связанных с семантическим анализатором

### 3.3 Разбиение на модули

Работать с большими проектами в разы удобнее и эффективнее при грамотном разбиении на модули. В экосистеме языка Rust такие модули именуются крейтами (англ. crates). На рисунке 7 представлено разбиение на модули

проекта Kodept. При этом стрелки указывают на зависимость одного модуля от другого.

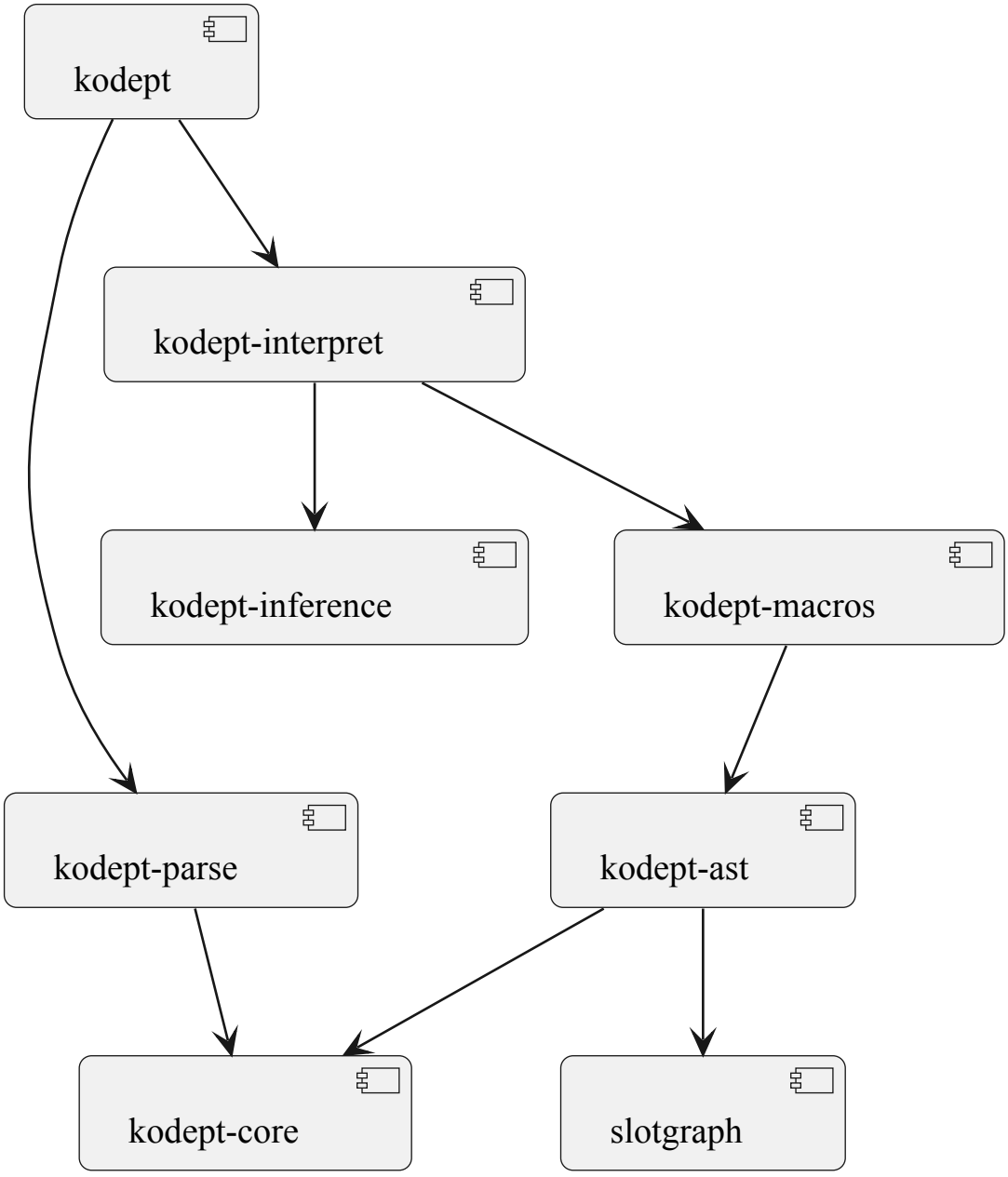


Рисунок 7 – Диаграмма иерархии модулей в исходном коде компилятора Kodept

В рамках этой работы внимание будет сконцентрировано вокруг модулей `kodept-ast`, `kodept-interpret` и `kodept-inference`, так как именно с помощью них решаются поставленные задачи. Однако, дадим краткое описание остальных модулей:

– `kodept-core` отвечает за определение основных структур данных, необходимых в остальных частях приложения, в частности, в нем определена структура *дерева разбора*,
– `kodept-parse` отвечает за лексический и синтаксический анализ, определяя набор синтаксических анализаторов (парсеров), которые генерируют дерево разбора,
– `kodept-macros` нужен для работы с AST и создания диагностик,
– `kodept` является корневым модулем и обеспечивает запуск стадий компилятора для входных файлов,
– `slotgraph` содержит реализацию графа с использованием *генеративной арены*.

Дерево разбора (англ. raw lexem tree, RLT) - подробное синтаксическое дерево, включающее в себя полную информацию об исходном коде. Необходимо для восстановления конкретной точки в программе при создании диагностики. В процессе работы программы трансформируется в AST, где каждый узел RLT соответствует определенному узлу AST.

Генеративная арена - специальный вид контейнера для объектов в программировании, в котором для каждого объекта присваивается не только уникальный идентификатор, но и поколение. Благодаря этому решается проблема утечек памяти: когда удаляется объект из обычной арены, то его место ничем не может быть занято, чтобы не нарушить идентификатор. В то же время в генеративной арене место из-под удалённого объекта может быть переиспользовано с помощью изменения поколения.

В модуле `kodept-ast` определена структура AST и принципы его хранения. Рассмотрим некоторые технические решения, реализованные при проектировании этого модуля.

## 3.4 Организация хранения абстрактного синтаксического дерева

Обычно абстрактное синтаксическое дерево реализовано в программе в виде вложенных друг в друга структур с данными, описывающими тот или иной синтаксис языка. Например, внутреннее представление функции с набором именованных параметров реализовано композицией структур (Рис. 8).

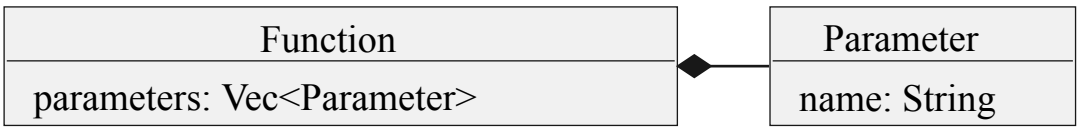


Рисунок 8 – Диаграмма классов, представляющих собой узел функции в AST

Основным процессом при работе семантического анализа является обход абстрактного синтаксического дерева с целью применения различных алгоритмов. Распространено использование так называемого шаблона посетитель (англ. visitor). Это специальная структура данных, содержащая алгоритмы для обработки каждой вершины дерева.

У такого подхода есть несколько минусов: 1) обход совершается в рекурсивном стиле, а это значит, что компилятор не способен обрабатывать большие исходные программы, 2) сложно поддерживать, так как необходимо изменить или добавить необходимые функции в код каждого посетителя при изменении вида AST,

В итоге был использован другой подход. Вместо хранения всех структур «вложенными» друг в друга (Рис. 8), было решено применить «графовый» метод. Суть его заключается в создании структуры-контейнера `SyntaxTree`. Все исходные структуры переписываются так, что из них убираются любые вложенные структуры, кроме базовых полей (имя и прочее). Также добавляется специальное поле `id`. Диаграмма классов после преобразования представлена на рисунке 9.

Такая композиция позволяет хранить все объекты AST в одном месте, линейно. Кроме того, реализация обхода дерева перестаёт быть рекурсивной и является реализацией алгоритма обхода в глубину. Для сохранения взаимосвязи между структурами, вводятся методы доступа. На рисунке 9 таким методом будет `parameters`.

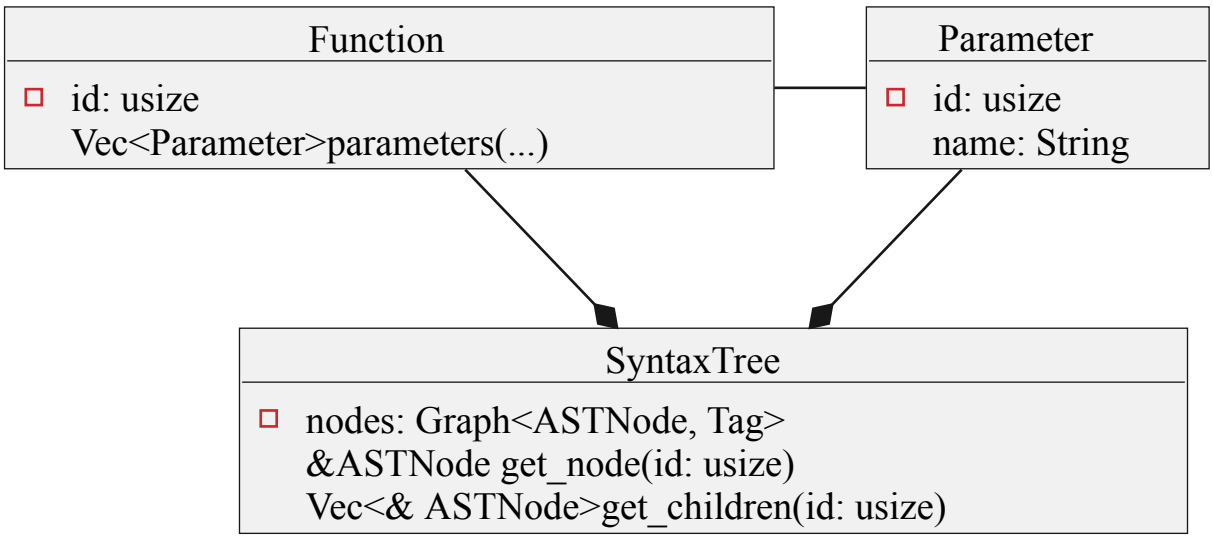


Рисунок 9 – Диаграмма классов после введённого преобразования

Рассмотрим подробнее структуру AST на рисунке А.1. В вершинах дерева расположены имена соответствующих структур данных вместе с идентификатором. Формат записи следующий: FileDecl $[\underset{\text{имя}}{\quad 1 \quad} \text{v} \underset{\text{идентификатор}}{\quad 1 \quad}]\underset{\text{поколение}}{}$. У всех вершин поколение равно единице, так дерево не модифицировалось. На некоторых рёбрах дерева расположились теги - маркеры, с помощью которых можно различить одинаковые по имени, но разные по семантике структуры. Например, обе вершины 11 и 12 являются ссылкой, но 11 - телом, а 12 - аргументом.

### 3.5 Реализация алгоритма $\mathcal{W}_c$

Реализация механизма вывода типов расположена в модуле `kodept-inference`. В нём также определены структуры для объектов, рассмотренных в разделе 2.4. Согласно рисунку 7, этот модуль не зависит от остальных. Достигается это тем, что часть AST конвертируется в эквивалентную структуру `Language` из модуля `kodept-inference`, представляющей собой терм (2.5). Затем для неё выводится тип, используя методы этого же модуля.

В процессе конвертации используется информация из *дерева областей видимости* (англ. scope tree) для правильного определения всех ссылок. Под ссылкой будем понимать идентификатор, ссылающийся на другой элемент исходного кода. Например, в листинге А.1, ссылкой является переменная `x` в записи `f(g(x))`.

### 3.5.1 Анализ областей видимости

Задачей этого анализа является разбиение AST на области видимости (англ. scopes), при котором строится так называемое дерево областей видимости. Вершинами дерева является структура `Scope`, представленная на рисунке 10. Принцип формирования дерева прост - обходится каждая вершина AST, если она является одной из вершин, которые образуют область видимости, то в дерево добавляется новый объект `Scope`, содержащий идентификатор текущей вершины AST и её имя. По мере обхода, заполняются ассоциативные массивы для типов (`types`) и переменных (`variables`) текущей области видимости. Благодаря им можно узнать, на что ссылалась та или иная ссылка с помощью методов `get_var` и `get_type`.

| Scope |
| --- |
| start_from_id: usize<br>name: Option<String><br>types: HashMap<String, MonomorphicType><br>variables: HashMap<String, usize><br>Option<String>get_var(name: String)<br>Option<MonomorphicType>get_type(name: String) |

Рисунок 10 – UML-диаграмма класса, представляющего вершину в дереве областей видимости

В список вершин, которые могут образовать область видимости, входят: `ModDecl`, `StructDecl`, `EnumDecl`, `AbstFnDecl`, `BodyFnDecl`, `FileDecl`, `Exprs`, `Lambda`, `IfExpr`.

Рассмотрим образование scope tree (Рис. 11) на примере программы с листинга А.1. Рисунок А.1 является изображением абстрактного синтаксического дерева для этой программы. В нём присутствует 7 вершин, образующих область видимости, однако в действительности областей 6, так как вершина с идентификатором 17 не имеет дочерних вершин.

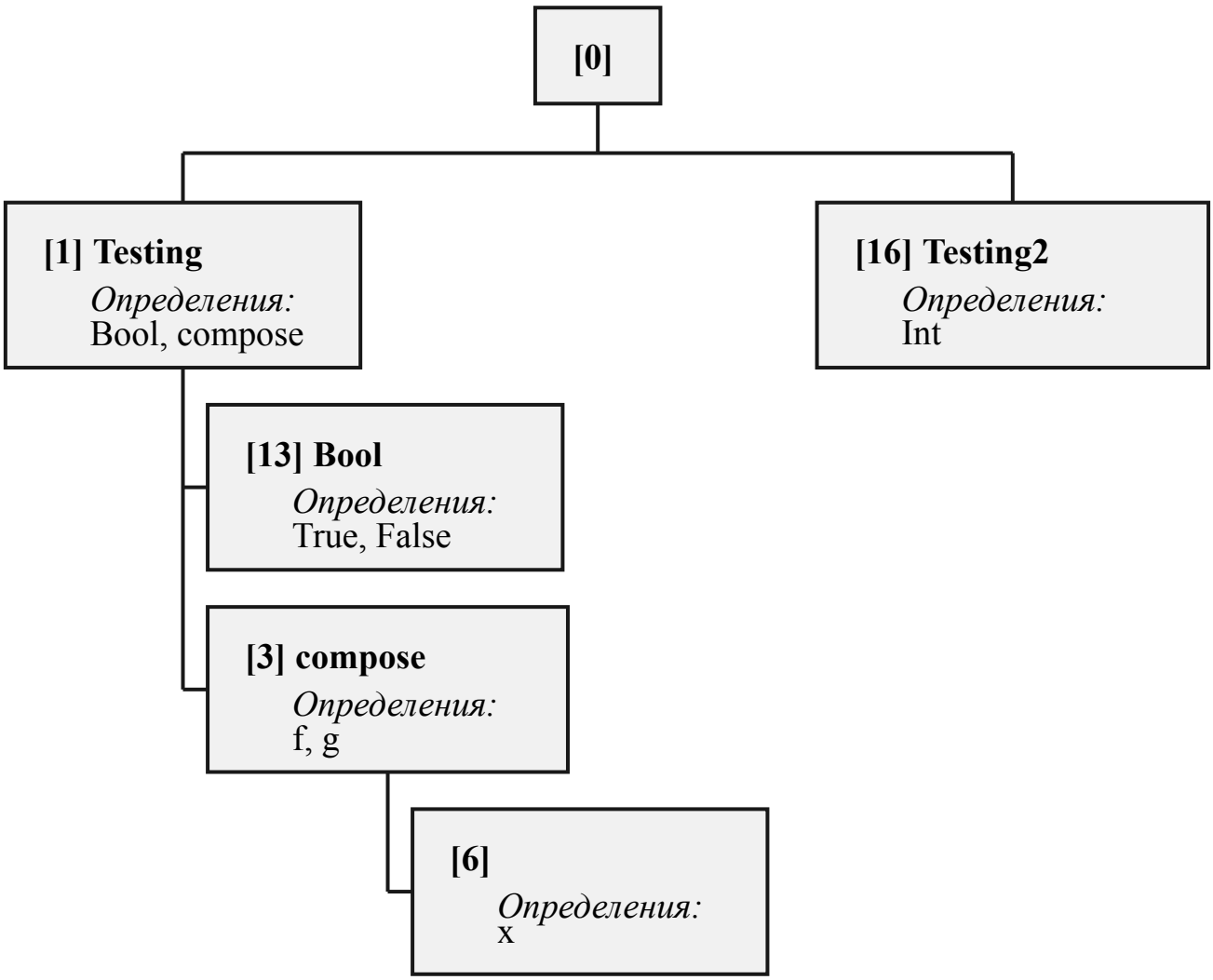


Рисунок 11 – Дерево областей видимости; в каждом элементе дерева в квадратных скобках указывается идентификатор вершины, после - имя области видимости

### 3.5.2 Преобразование AST в термы

Определенный в разделе 2.7, алгоритм вывода типов $\mathcal{W}_c$ не может работать со структурой абстрактного синтаксического дерева. Для того чтобы проверить типы в программе, необходимо преобразовать AST в подходящую для алгоритма структуру. При этом преобразуются не всё дерево, а лишь некоторая его часть. Преобразование происходит с помощью функции `convert(...)` по правилам, которые указаны ниже. Будем использовать нотацию $\underline{e}$ для результата вызова функции `convert(e)`.

Правило преобразования функции с телом `BodyFnDecl [name, expr, parameters ]`, где `parameters` - параметры функции, `expr` - тело функции:

$$\begin{cases} \underline{expr} \text{ если параметров нет,} \\ \lambda \underline{p1}.\lambda \underline{p2}.\cdots.\underline{expr}, \end{cases} \tag{3.32}$$

где `p1, p2` $\in$ `parameters`.

Правило для преобразования применения `Appl [ expr, parameters ]`:

$$\begin{cases} \underline{expr} \text{ если параметров нет,} \\ ((\underline{expr}(\underline{p1}))(\underline{p2}))\ldots, \end{cases} \tag{3.33}$$

где `p1, p2` $\in$ `parameters`.

Правило для преобразования набора выражений `Exprs [ items ]`:

$$\begin{cases} () & \text{если список } \texttt{items} \text{ пуст,} \\ \text{let } n = \underline{i1} \text{ in } (\underline{i2}) & \text{если } \underline{i1} \neq \text{ let } x = e_1 \text{ in } e_2, \\ \text{let } x = e_1 \text{ in } \underline{i2} & \text{иначе,} \end{cases} \tag{3.34}$$

где `i1, i2` $\in$ `items`,

`n` - имя элемента `i1` (если есть, иначе - идентификатор вершины).

Правило для преобразования объявления инициализированной переменной `InitVar [ name, expr ]`:

$$\text{let } name = \underline{expr} \text{ in } name \tag{3.35}$$

Правило для преобразования условного выражения `IfExpr [ condition, body, elifs, else ]`, где `elifs` - список с элементами вида `ElifExpr [ condition, body ]`, `else` имеет вид `ElseExpr [ body ]`:

$$\text{if } \underline{condition} \text{ then } \underline{body} \text{ otherwise } e_1, \tag{3.36}$$

где $e_i =$ if $\underline{efi.condition}$ then $\underline{efi.body}$ otherwise $e_{i+1}$,

$e_n = \underline{else}$,

`efi` $\in$ `elifs`.

Правило для преобразования числовых литералов тривиально и представляет собой использование терма :num:. Преобразование кортежа - преобразование каждого элемента внутри кортежа. Правило для ссылки `Ref [ name]` : $name$.

Правило для преобразования лямбда-функции `Lambda [ binds, expr ]`, где `binds` - список параметров лямбда-функции:

$$\begin{cases} \underline{expr} \text{ если параметров нет,} \\ \lambda \underline{b1}.\lambda \underline{b2}.\cdots.\underline{expr}, \end{cases} \tag{3.37}$$

где `b1, b2` $\in$ `binds`.

Вывод типов в программе проводится отдельно для каждого определения функции с телом. При этом к структуре функции применяется функция `convert` с последующим получением терма $e$. Затем производится первый этап алгоритма $\mathcal{W}_c$ - к терму $e$ применяются правила вывода в форме, определённой в разделе 2.5.1. Таким образом имеется набор предположений $\mathcal{A}$ (2.16) и набор ограничений $\mathcal{C}$. Для каждого элемента $x : \tau$ набора предположений рекурсивно запускается отдельный процесс преобразования узла AST, определяющего $x$, а результат процесса записывается в контекст $\Gamma$. Благодаря этому второй этап алгоритма $\mathcal{W}_c$ имеет полностью определенный набор внешних определений $\Gamma$.

После решения ограничений с помощью алгоритма 2.29, будет получен тип $\tau$ для выражения $e$. Этот тип обобщается (2.10) и выступает в качестве выведенного для рассматриваемой функции.

Рассмотрим работу функции `convert` на примере листинга А.1. В нём содержится одна функция `convert`, представляющая собой одно лямбда-выражение. Последовательно применяя правила преобразования, можно получить следующий терм для этой функции: $\lambda f.\lambda g.\lambda x.f(g(x))$.

# 4 Сборка и тестирование

В экосистеме языка Rust принято использовать Cargo [18] для сборки приложения, управления зависимостями и тестирования. Для сборки компилятора Kodept тоже необходимо использовать этот инструмент. На данный момент используется превью-версия 1.80.0 компилятора Rust для доступа к нестабильным функциям языка, что тоже необходимо учитывать при сборке. Собрать проект можно выполнив следующую команду в командной строке:

```
cargo build --release
```

Пакетный менеджер Cargo скачает необходимые зависимости и соберёт программу. Приложение выполняется кросс-платформенно, его работа была протестирована на ОС Linux и на ОС Windows. Дополнительных требований к получаемому исполняемому файлу не предоставляется.

При разработке приложения не обойтись без тестирования. Оно помогает выявить различные ошибки и исправить их. Популярным вариантом тестирования является модульное тестирование (англ. unit тестирование). Unit тест - функция или набор функций, которые проверяют корректность работы отдельной нетривиальной части программы.

В ходе разработки компилятора были написаны модульные тесты, они покрывают большое количество кода и успешно выполняются. Для их запуска требуется Cargo с использованием следующей команды:

```
cargo test --all --all-features --no-fail-fast
```

Компилятор Kodept можно собрать с поддержкой профилирования использованной оперативной памяти. Для этого можно воспользоваться отдельным механизмом Cargo - особенность (англ. feature). Профилирование памяти помогает узнать использование оперативной памяти в разные моменты программы. Так, например, для исходного кода с листинга А.1, на рисунке 12 показаны результаты профилирования используемой памяти компилятора. Первые три строки вывода занимают логи о прохождении этапов компиляции. Затем приведена информация о профилировании: за всё время работы было использовано 320Кб, максимальный размер - 61Кб, в момент завершения выполнения - 34Кб.


Рисунок 12 – Скриншот вывода компилятора Kodept при запуске с опцией профилирования

Рассмотрим теперь непосредственно работу вывода типов в программе. Для исходного кода с листинга А.1 в разделе 3.5.2 было сказано, что образованный из функции `compose` терм $e$ выглядит так: $\lambda f.\lambda g.\lambda x.f(g(x))$. Если воспользоваться алгоритмом $\mathcal{W}_c$, приведёнными в разделе 2.7, то тип $\tau$ терма $e$ будет следующим: $\forall\{\alpha_1, \alpha_2, \alpha_3\}.(\alpha_2 \to \alpha_3) \to (\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_3)$. Действительно, тип этой функции, определённый компилятором совпадает с $\tau$ с точностью до имен переменных типа (Рис. 13).


Рисунок 13 – Скриншот вывода компилятора Kodept с типом функции compose

# ЗАКЛЮЧЕНИЕ

В результате выполнения данной работы был реализован механизм вывода типов для языка программирования Kodept. Для этого была сформирована система типов на основе системы типов Хиндли-Милнера, а также алгоритм для вывода типов $\mathcal{W}_c$. Показано, что программная реализация в виде компилятора может выводить типы в предоставляемых исходных файлах на языке Kodept.

В ходе работы рассмотрена архитектура компилятора, а также методы, применяемые при работе с абстрактным синтаксическим деревом. Рассмотрены программные модули, отвечающие как за сам вывод типов, так и за смежные с ним части компилятора. Успешно решены все поставленные задачи, а именно:

1) проанализированы некоторые системы типов в современных языках программирования,
2) приведена модификация системы типов Хиндли-Милнера,
3) рассмотрен алгоритм вывода типов согласно выбранной системы типов,
4) реализован модуль в компиляторе языка Kodept для семантического анализа,
5) выполнено тестирование компилятора на правильность вывода типов.

На данной работе разработка как языка программирования Kodept, так и компилятора для него не завершается. Поэтому в качестве дальнейших планов можно выделить следующие:

– дальнейшая доработка и усложнение системы типов,
– расширение семантического анализатора,
– доработка компилятора: добавление поддержки многопоточного выполнения, реализация следующих стадий компиляции, таких как генерация промежуточного представления и генерация машинного кода.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Голованов Вячеслав. Насколько близко компьютеры подошли к автоматическому построению математических рассуждений? [Электронный ресурс]. 2020. (Дата обращения 22.04.2024). URL: https://habr.com/ru/articles/519368/.

2. Milner Robin. A theory of type polymorphism in programming // Journal of computer and system sciences 17. 1978. С. 348–375.

3. Свиридов Сергей. Теория типов [Электронный ресурс]. 2023. (Дата обращения 19.04.2024). URL: https://habr.com/ru/articles/758542/.

4. Груздев Денис. Ликбез по типизации в языках программирования [Электронный ресурс]. 2012. (Дата обращения 18.04.2024). URL: https://habr.com/ru/articles/161205/.

5. Why is Python a dynamic language and also a strongly typed language [Электронный ресурс]. (Дата обращения 21.04.2024). URL: https://wiki.python.org/moin/Why is Python a dynamic language and also a strongly typed language.

6. C Reference [Электронный ресурс]. (Дата обращения 21.04.2024). URL: https://en.cppreference.com/w/c.

7. Объектно ориентированное программирование на Си без плюсов [Электронный ресурс]. (Дата обращения 20.04.2024). URL: https://habr.com/ru/articles/568588/.

8. Simion Dănuț-Octavian. Using Java in Business Applications // WSEAS Press. 2010. 01.

9. Benjamin J Evans David Flanagan. Java in a Nutshell. 7th Ed. O'Reilly Media, Inc., 2018.

10. Deutsch Harry, Marshall Oliver. Alonzo Church // The Stanford Encyclopedia of Philosophy / под ред. Edward N. Zalta, Uri Nodelman. Metaphysics Research Lab, Stanford University, 2023.

11. Barendregt H.P. The lambda calculus: its syntax and semantics. Studies in logic and the foundations of mathematics; v. 103. 621 с.

12. Barendregt Henk (Hendrik). Lambda Calculi with Types. 1992. 01. T. 2. С. 117–309.

13. Urban Christian, Nipkow Tobias. Nominal verification of algorithm W // From Semantics to Computer Science. Essays in Honour of Gilles Kahn / под ред. G. Huet, J.-J. Lévy, G. Plotkin. Cambridge University Press, 2009. С. 363–382.

14. badcasedaily1. Как работает управление памятью в Rust без сборщика мусора [Электронный ресурс]. (Дата обращения 15.04.2024). URL: https://habr.com/ru/companies/otus/articles/787362/.

15. Matsakis Niko. Polonius: Either Borrower or Lender Be, but Responsibly // Rust Belt Rust Conference. 2019.

16. GraphViz Documentation [Электронный ресурс]. 2022. (Дата обращения 24.04.2024). URL: https://www.graphviz.org/documentation/.

17. Nystrom Robert. Crafting Interpreters [Электронный ресурс]. (Дата обращения 23.03.2023). URL: https://craftinginterpreters.com.

18. The Cargo Book. (Дата обращения 26.05.2024). URL: https://doc.rust-lang.org/cargo/.

# ПРИЛОЖЕНИЕ А

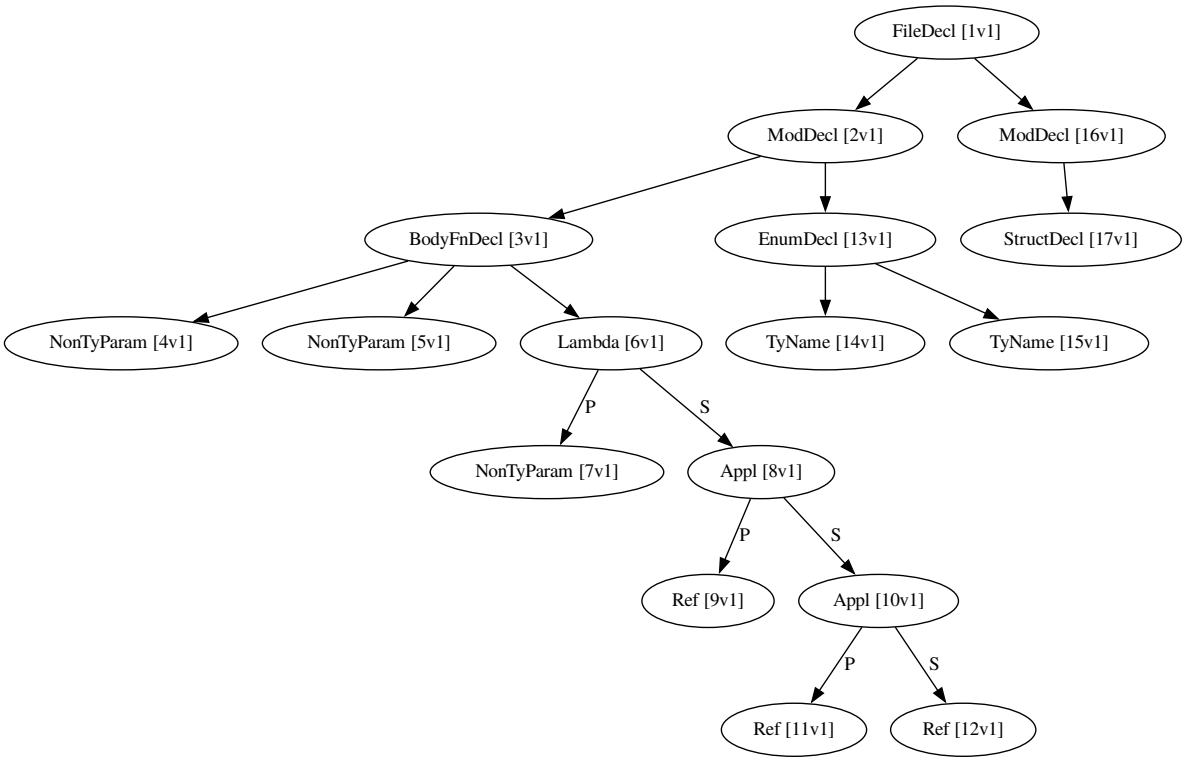


Рисунок А.1 – Изображение структуры абстрактного синтаксического дерева

Листинг А.1 – Исходная программа на языке Kodept

```
1 module Testing {
2     fun compose(f, g) => \x => f(g(x))
3
4     enum struct Bool { True, False }
5 }
6
7 module Testing2 {
8     struct Int
9 }
```

# ПРИЛОЖЕНИЕ Б

Листинг Б.1 – Имена всех элементов, составляющих абстрактное синтаксическое дерево

```
1    File, // root element
2    Module, // module name rest
3    Struct, // struct name(params) rest
4    Enum, // enum name rest
5    TypedParameter, // name: type
6    UntypedParameter, // name
7    Variable, // val name: type
8    InitializedVar, // val name: type = expr
9    BodiedFunction, // fun name(params) => expr
10   ExpressionBlock, //  expr1; expr2; ...
11   Application, // expr(expr)
12   Lambda, // \binds => expr
13   Reference, // name
14   Access, // expr.expr
15   Number, // number literal
16   Char, // char literal
17   String, // string literal
18   Tuple, // (expr1, expr2, ...)
19   If, // if expr => expr другие ветки
20   Elif, // elif expr => expr
21   Else, // else expr
22   Binary, // binary operator: +, -, *, /, %, ^
23   Unary, // unary operator: -, +, !, 
24   AbstractFunction, // abstract fun name(params): type
25   ProdType, // (type1, type2, ...)
```