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

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)

УТВЕРЖДАЮ

Заведующий кафедрой *РК-6*
индекс

_____ *А.П. Карпенко*

«____» _____ 2024 г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

Студент группы: РК6-85Б

Никитин Владимир Леонидович

(фамилия, имя, отчество)

Тема выпускной квалификационной работы: Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера

Источник тематики (кафедра, предприятие, НИР): кафедра

Тема выпускной квалификационной работы утверждена распоряжением по факультету РК № _____ от «____» _____ 2024 г.

Техническое задание

Часть 1. *Анализ актуальности.*

Должен быть выполнен анализ существующих система систем типов в современных языках программирования. Также должна быть обоснована актуальность разработки в целом.

Часть 2. *Математическая постановка задачи, разработка архитектуры программной реализации, программная реализация.*

Должна быть описана система типов и реализована в языке программирования Kodept в качестве механизма вывода типов

Часть 3. *Проведение тестирования.*

Должно быть проведено тестирование разработанной в ходе работы над курсовым проектом программы.

Оформление выпускной квалификационной работы:

Расчетно-пояснительная записка на 52 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):

количество: 11 рис., 4 табл., 0 источн.

Дата выдачи задания «*01*» октября 2024 г.

В соответствии с учебным планом выпускную квалификационную работу выполнить в полном объёме в срок до «____» _____ 2024 г.

Студент _____ (подпись, дата) Никитин В.Л. (ФИО)

Руководитель выпускной квалификационной работы _____ (подпись, дата) Соколов А.П. (ФИО)

Примечание. Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

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 РК
КАФЕДРА РК-6
ГРУППА РК6-85Б

УТВЕРЖДАЮ
Заведующий кафедрой *РК-6*
индекс
_____ *А.П. Карпенко*
«____» _____ 2024 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Студент группы: РК6-85Б
Никитин Владимир Леонидович
(фамилия, имя, отчество)

Тема выпускной квалификационной работы: Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера

№ п/п	Наименование этапов выпускной квалификационной работы	Сроки выполнения этапов		Отметка о выполнении	
		план	*факт*	Должность	ФИО, подпись
1.	Задание на выполнение работы. Формулировка проблемы, цели и задач работы	18.02.2024	*18.02.2024*	Руководитель ВКР	Соколов А.П.
2.	1 часть: аналитический обзор литературы	18.02.2024	*31.03.2024*	Руководитель ВКР	Соколов А.П.
3.	Утверждение окончательных формулировок решаемой проблемы, цели работы и перечня задач	28.02.2024	*28.02.2024*	Заведующий кафедрой	А.П. Карпенко

№ п/п	Наименование этапов выпускной квалификационной работы	Сроки выполнения этапов		Отметка о выполнении	
		план	***факт***	**Должность**	**ФИО, подпись**
4.	2 часть: математическая постановка задачи, разработка архитектуру программной реализации, программная реализация	31.03.2024	*31.03.2024*	Руководитель ВКР	Соколов А.П.
5.	3 часть: проведение вычислительных экспериментов, отладка и тестирование	30.04.2024	*30.04.2024*	Руководитель ВКР	Соколов А.П.
6.	1-я редакция работы	31.05.2024	*31.05.2024*	Руководитель ВКР	Соколов А.П.
7.	Подготовка доклада и презентации	17.06.2024	*17.06.2024*		
8.	Заключение руководителя	15.06.2024	*15.06.2024*	Руководитель ВКР	Соколов А.П.
9.	Допуск работы к защите на ГЭК	15.06.2024	*15.06.2024*	Нормоконтролер	С.В. Грошев
10.	Внешняя рецензия	12.06.2024	*12.06.2024*		
11.	Защита работы на ГЭК	19.06.2024	*19.06.2024*		

Студент _____ Никитин В.Л. Руководитель ВКР_____ Соколов А.П.
подпись, дата ФИО подпись, дата ФИО

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)

НАПРАВЛЕНИЕ
НА ЗАЩИТУ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Председателю
Государственной Экзаменационной Комиссии № _____

факультета «Робототехника и комплексная автоматизация» МГТУ им. Н.Э. Баумана

Направляется студент *Фамилия Имя Отчество* группы *РК6-81Б*

на защиту выпускной квалификационной работы *Тема*

Декан факультета _____ подпись декана _____ «11» июня 2020 г.

Справка об успеваемости

Студент *Фамилия Имя Отчество* за время пребывания в МГТУ имени Н.Э. Баумана с *2017* г. по *2020* г. полностью выполнил учебный план со следующими оценками: отлично – *[процент]* %, хорошо – *[процент]* %, удовлетворительно – *[процент]* %.

Инспектор деканата _____ от руки _____

Отзыв руководителя выпускной квалификационной работы

Студент Фамилия И.О. в процессе выполнения ВКР проявил себя как … Результаты, полученные в процессе реализации задания, позволили сделать вывод о … целесообразности/нецелесообразности выбранных путей решения поставленной задачи, … невозможности применения … Работа выполнена автором самостоятельно, в полном объёме, в полном соответствии с заданием и календарным планом. Несмотря на сделанные замечания студент достоин «отличной» оценки… и присвоения звания бакалавр техники и технологий по направлению «Информатика и вычислительная техника».

запросить у руководителя.

запросить у руководителя

Руководитель ВКР _____ А.П. Соколов «____» _____ 2020 г.
(подпись) (ФИО) (дата)

Студент _____ И.О. Фамилия «____» _____ 2020 г.
(подпись) (ФИО) (дата)

подписать

РЕФЕРАТ

выпускная квалификационная работа: 52 с., 11 рис., 4 табл., 0 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. В последнее время все больше и больше языков почерпывают что-то из этой области. Применение мощной системы типов позволяет зачастую снизить количество ошибок, возникающих при разработке.

Тип работы: выпускная квалификационная работа.

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

Объект исследования: система типов.

Основная задача, на решение которой направлена работа: реализация алгоритма вывода типов на основе выбранной системы типов.

Цели работы: реализация системы вывода и проверки типов

В результате выполнения работы: 1) спроектировано представление абстрактного синтаксического дерева в компиляторе; 2) реализован семантический анализатор; 3) показано, что компилятор успешно может вывести тип функции

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.

Одной из проблем при программировании является различного рода ошибки.

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 [**?**]. 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.

Listing 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. В.1). Одной из них является семантический анализ. В него входят анализ областей видимости объявлений, *проверка (и вывод) типов* и др.

Проверкой типов называют процесс, когда тем или иным образом проверяется правильность типа выражения согласно системе типов языка. В ходе научно-исследовательской работы была выбрана система типов Хиндли-Милнера. Алгоритм W [**?**], реализующий ее, позволяет также проводить вывод типов.

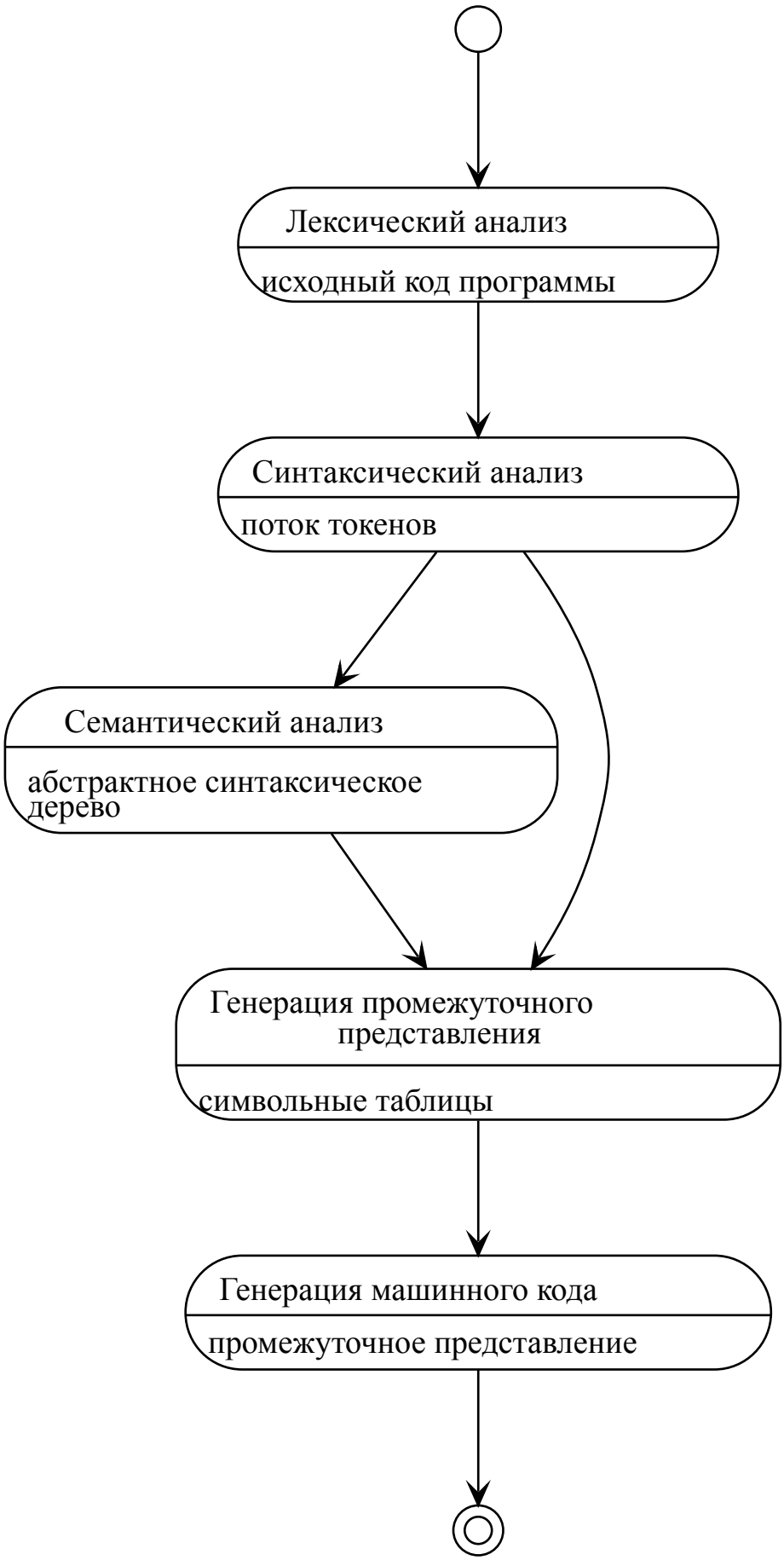

Рисунок В.1 – Схема состояний работы компилятора

Целью курсового проекта является реализация механизма вывода и проверки типов для языка программирования Kodept в качестве его дальнейшего развития.

Улучшить качество новых языков программирования, и применить ее на примере Kodept. Актуальность: статическая типизация в языках программирования важна. Проблема: уменьшение багов.

1. Постановка задачи

1.1. Концептуальная постановка задачи

Объект разработки: система типов

Цель: реализовать систему вывода и проверки типов

Задачи:

1) спроектировать представления AST в компиляторе,
2) реализовать анализатор областей видимости,
3) написать алгоритм для вывода типов.

1.2. Математическая постановка задачи

Поставить задачу

2. Математическая постановка задачи

Не уверен насчет названия главы, может даже переименовать в вычислительный метод

2.1. Форма Бэкуса-Наура

Целая секция для просто определения BNF. звучит плохо...

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: $A := B$, 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 $:=$ 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 $K := * \mid * \to *$, множеством возможных значений является $\{*, * \to *, (* \to *) \to *, \ldots\}$.

2.2. Теория типов

В разделе представлена информация о специальном разделе математики - теории типов [**?**]. 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. Поэтому 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 [**?**] (таблица 2), 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 типизация.

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 Γ - контекст,

P - предположение.

Всего существует 6 видов суждений:

$\Gamma \vdash$	Γ - верный контекст,
$\Gamma \vdash \tau$	τ - тип в контексте,
$\Gamma \vdash x : \tau$	терм x имеет тип τ в контексте,
$\vdash \Gamma = \Delta$	контексты Γ и Δ равны,
$\Gamma \vdash \tau_1 = \tau_2$	типы τ_1 и τ_2 в контексте равны,
$\Gamma \vdash x_1 = x_2 : \tau$	терм x_1 равен терму x_2 с типом τ.

Таблица 2 Изоморфизм Карри-Ховарда

Логическое высказывание	**Язык программирования**
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, $\bot$
$\neg F$	$A \to \bot$

Тип T обитаем (англ. inhabitat), если выполняется следующее: $\exists t : \Gamma \vdash t : T$ - найдется терм t, такой, что в контексте Γ 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 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 Γ 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* [**?**]. Это влияет на то, в какой момент в программе происходит проверка соответствия типов. В динамических системах - во время исполнения программы, 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 [**?**]. 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 ра-

боты 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. **дополнить?**

2.3.1. Система типов C

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

Типом в языке C является интерпретация набора байт, составляющих объект [**?**]. Все типы бывают двух видов: базовые и производные (Рисунок 2). 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).

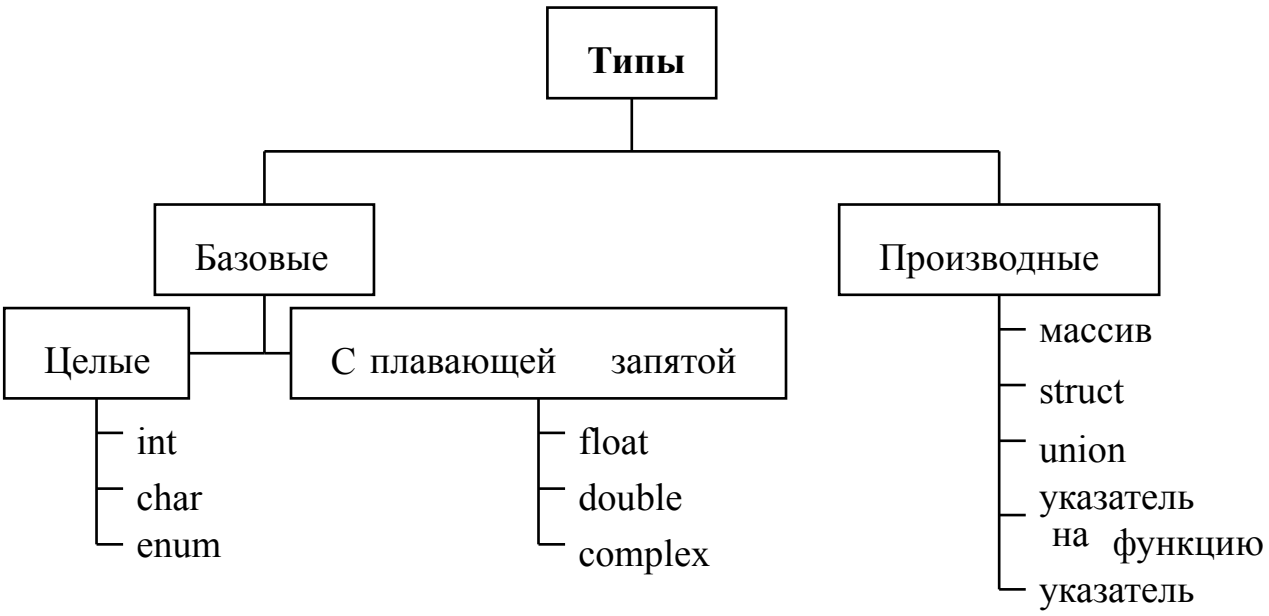

Рисунок 2 – Схематичное изображение типов в C

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

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

Достоинства:

– C 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 (2.2).

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

Listing 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.

Систему типов, применяемую в этом языке можно охарактеризовать как статическую, сильную и явную с возможностью введения неявно типизированных выражений [**?**]. Java создавалась под сильным влиянием идей объектно-ориентированного программирования, поэтому она включает классы, интерфейсы, 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. 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 α-эквивалентность, β-редукцию и η-преобразование [**?**].

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 [**?**]. Это обобщение называется лямбда-кубом и туда входят восемь типизированных лямбда-исчислений (рис. 3). Также на лямбда-кубе отмечены зависимости между его элементами так, что ребро $\rightarrow$ обозначает отношение включения ($\subseteq$).

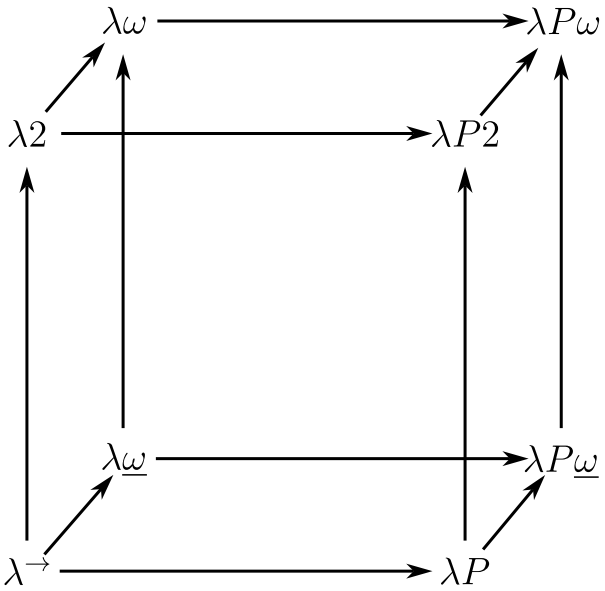

Рисунок 3 – Графическое изображение лямбда-куба

С помощью куба описываются четыре основных зависимости между типами и термами:

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

– типы зависят от термов (λP).

Согласно рис. 3, система $\lambda^{\rightarrow}$ (просто типизованное лямбда-исчисление) является самой простейшей и включена во все остальные. Рассмотрим её подробнее. В множество возможных типов входит переменная типа α и функциональный тип $\tau_1 \rightarrow \tau_2$. Таким образом, $\tau \coloneqq \alpha \mid \tau_1 \rightarrow \tau_2$. 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 C, за исключением пользовательских типов и массивов.

В систему $\lambda 2$ (полиморфное лямбда-исчисление), по сравнению с предыдущей системой, добавляется так называемый полиморфный тип $\sigma \coloneqq \forall\alpha.\tau \mid \tau$. Таким образом, терм может зависеть от конкретного типа τ: $f : \forall\alpha.\alpha \rightarrow \alpha \implies f(\tau) : \tau \rightarrow \tau$. 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 α в тип α: $f : \alpha \rightarrow \alpha$. 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):

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

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 $f : \tau \rightarrow *$, где $*$ - ранее рассмотренная категория типов. Тогда f является конструктором, а $(\lambda a : \tau.f(a)) : *$, где a - терм с типом τ. 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 следующей таблицы:

Таблица 3 Наличие зависимостей между категорией типов ($*$) и категорией видов ($\Box$) в системах типов, описанных в лямбда-кубе.

Система типов	**Используемые правила**
$\lambda^{\rightarrow}$	$(*, *)$
$\lambda 2$	$(*, *)$, $(\Box, *)$
λP	$(*, *)$, $(*, \Box)$
$\lambda \underline{\omega}$	$(*, *)$, $(\Box, \Box)$
$\lambda \omega$	$(*, *)$, $(\Box, *)$, $(\Box, \Box)$
$\lambda P \underline{\omega}$	$(*, *)$, $(*, \Box)$, $(\Box, \Box)$
$\lambda P \omega$	$(*, *)$, $(\Box, *)$, $(*, \Box)$, $(\Box, \Box)$

Конструкция вида (s_1, s_2), где $s \in \{*, \Box\}$ 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, таким образом код становится самодокументируемым,
- 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. Далее приводится модификация системы типов Хиндли-Милнера.

2.4. Система типов Хиндли-Милнера

может по-другому назвать секцию, я использую своего рода свою систему типов

Для начала определим термы:

$$
\begin{aligned}
e_1, e_2, e_3 \coloneqq\ & x && \text{(2.5a)}\\
\mid\ & e_1(e_2) && \text{(2.5b)}\\
\mid\ & \lambda x : \tau . e_1 && \text{(2.5c)}\\
\mid\ & \text{let } x : \tau = e_2 \text{ in } e_2 && \text{(2.5d)}\\
\mid\ & \text{:num:} && \text{(2.5e)}\\
\mid\ & (e_1, e_2) && \text{(2.5f)}\\
\mid\ & \text{if } e_1 \text{ then } e_2 \text{ otherwise } e_3, && \text{(2.5g)}
\end{aligned}
$$

где e_1, e_2, e_3 - термы,
x - имя переменной (англ. binding),
:num: - числовой литерал,
τ - 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 безымянной, лямбда, функции.

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

(2.5f) конструкция необходима для создания пар из двух других термов,

В некоторых элементах к переменной x приписывается ограничение на тип τ. Это необходимо для того, чтобы явно указать необходимый тип, как это делается в других языках программирования (листинг 2.3).

Listing 2.3 – Явное указание типа аргумента в языке C.

```
1   void foo(int a) { }
```

Теперь можно ввести определение типа τ.

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

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

α - 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 программиро-

вания, 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.

Listing 2.4 – Определение обобщенной функции в C++

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

Далее введем следующие понятия: *контекст* Γ (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,

Σ - 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 τ,

$X \backslash Y$ - 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 α_i из подстановки $\mathcal{S}$ на τ_i в типе τ.

Будем называть β - уникальную переменную типа.

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

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

Правила вывода, разработанные Милнером [**?**], 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 τ_1 и τ_2, что $\mathcal{S}\tau_1 = \mathcal{S}\tau_2$. Для решения этой задачи существует алгоритм $\mathcal{U}$:

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

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 τ_1 и τ_2 должны быть унифицированы.

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

– Неявное ограничение на экземпляр ($\tau_1 \leqslant_{\mathcal{M}} \tau_2$) показывает, что тип τ должен быть конкретным экземпляром типа, полученного после обобщения типа τ_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, τ - тип. Сами правила представлены в подразделе 2.5.1 В отличие от контекста Γ, 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

Каждое правило записано одним выражением в том виде, который определен в разделе 2.2. Для каждого варианта терма определено собственное правило. Таким образом любой терм может быть типизирован.

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 $e = \text{let } x : \tau = e_1 \text{ in } e_2$:

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

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

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

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

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

Правило для варианта терма $e = \text{if } e_1 \text{ then } e_2 \text{ otherwise } e_3$:

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

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 (**добавить их в приложение**), 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.

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

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

Рассмотрим решение множества ограничений 2.26:

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

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++:

может лучше std::function<T(std::function<T(int)>)>

Listing 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 Γ и терм e, возвращает - подстановку $\mathcal{S}$ и тип терма τ.

Введём явное ограничение на экземпляр и для множеств $\mathcal{A}$ и Γ (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 4 представлена блок-схема рассматриваемого алгоритма. Из нее видно, что процесс применения правил вывода не зависит от использования контекста Γ, что позволяет типизировать любой терм, вне зависимости от внешних переменных. Для определения того, что все переменные из множества предположений определены, используется условие $dom(\mathcal{A}) \nsubseteq dom(\Gamma) = \{x \mid x : \tau \in \mathcal{A}\} \nsubseteq \{x \mid x : \sigma \in \Gamma\}$.

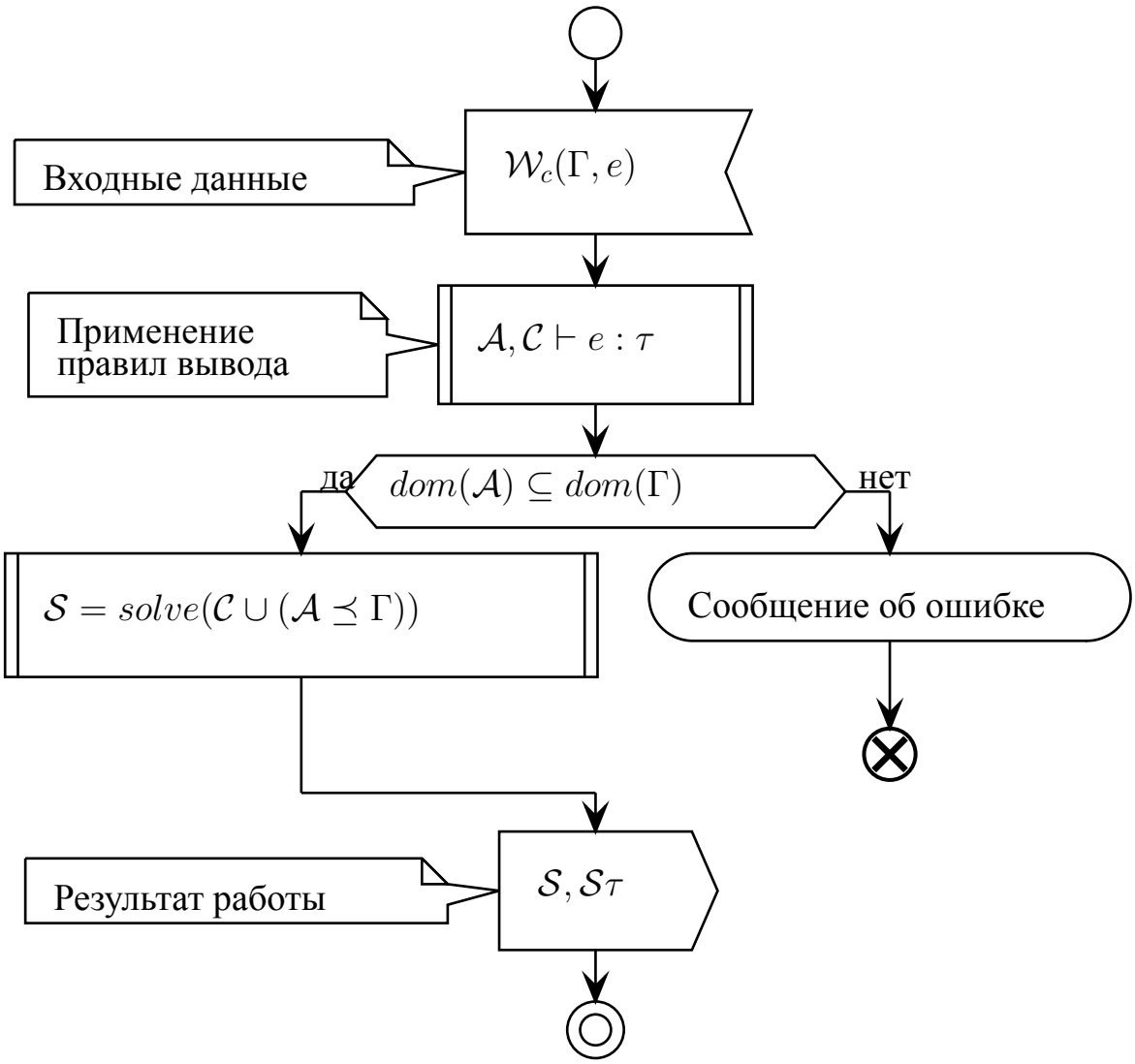

Рисунок 4 – Блок-схема алгоритма $\mathcal{W}_c$

3. Программная реализация

3.1. Архитектура

Одной из поставленных задач является реализация алгоритма вывода типов в компиляторе языка Kodept. Компилятор обычно представляет собой приложение командной строки (англ. CLI app). Также, в структуре программы почти любого компилятора можно выделить 3 основные части (рис. 5).

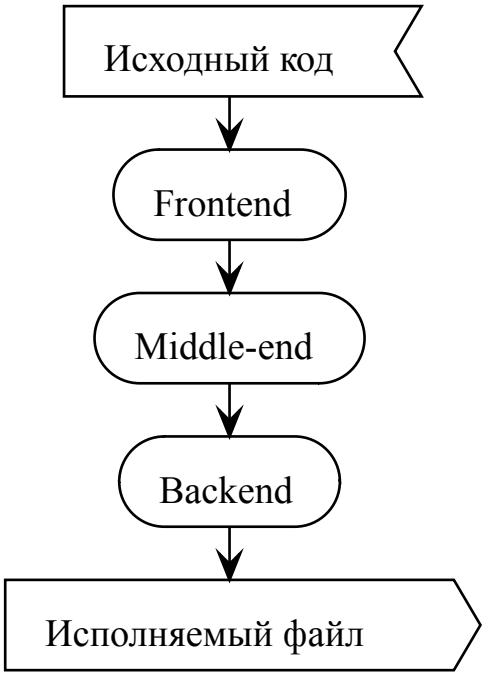

Рисунок 5 – Архитектура большинства современных компиляторов

Сопоставляя эти части с рис. В.1 можно прийти к такому выводу: 1) к frontend части относятся лексический и синтаксический анализ, 2) к middle-end части - семантический анализ и генерация промежуточного представления, 3) к backend части - генерация машинного кода. Разработка frontend части была завершена ещё до этой работы и не будет подробно раскрываться.

Проект разрабатывается с использованием языка программирования Rust. Этот язык предлагает надежный концепт управления памятью, не имея при этом сборщика мусора [**?**]. 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,
– система сборки создает нативный файл программы - его можно запустить, не имея на машине специальных сред выполнения.

Как уже было сказано, компилятор языка Kodept является консольным приложением, что повлияло на его варианты использования (Рисунок 6). Отдельным пользователем системы является сам разработчик компилятора. Ему необходим доступ к расширенной информации о работе процесса компиляции, поэтому существуют отдельные варианты использования, предназначенные специально для него. Важно понимать, что компилятор всё ещё находится в разработке, поэтому будет добавлено ещё больше вариантов использования. Реализация существующих вариантов выполнена в виде набора команд и флагов для интерфейса командной строки.

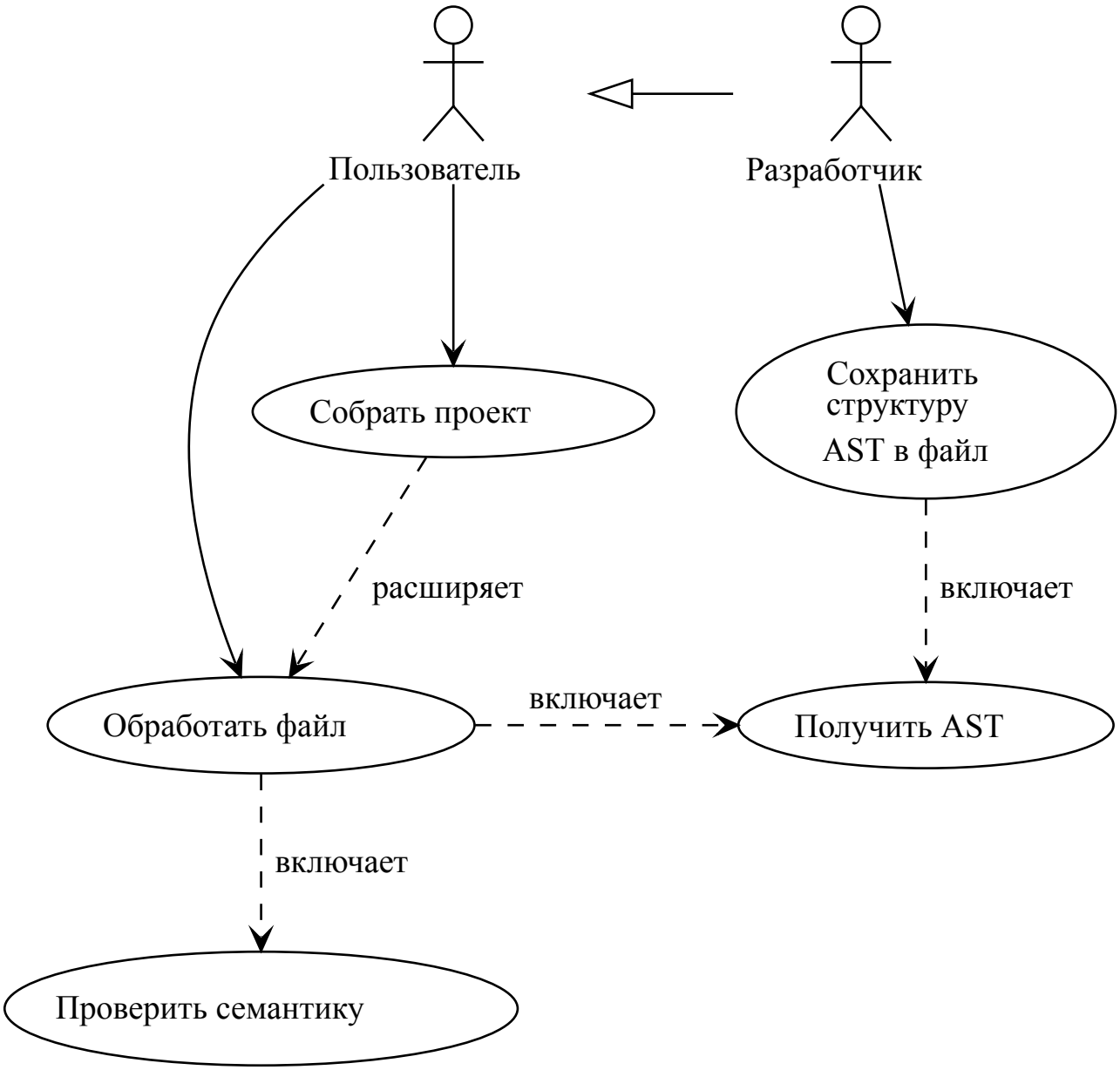

Рисунок 6 – Варианты использования компилятора языка Kodept

Рассмотрим интерфейс командной строки подробнее. На данный момент присутствует 2 основных команды - `graph` и `run`. С помощью первой можно выполнить первый этап компиляции и получить AST. Затем программа сохраняет его в файл в виде диаграммы на языке `dot`. Этот режим подразумевается для отладки AST и не выполняет преобразований над ним. Вторая же команда делает гораздо больше работы и необходима для выполнения всех этапов компиляции. Кроме команд, можно использовать различные опции компилятора. Полный список таких опций приведён в таблице 4.

Диагностика (диагностическое сообщение) - сообщение компилятора, выводимое в стандартный поток ошибок. Необходима для указания проблемы, ошибки или другой информации об исходном коде. Содержат непосредственно место в коде, уровень ошибки, а также дополнительное описание.

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

Таблица 4 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

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 сплошной линией выделено отношение модулей (от независимых к зависимым), пунктирной линией отмечена передача данных во

время работы программы (от начала к концу). Под диагностиками необходимо понимать сообщения компилятора об исходной программе.

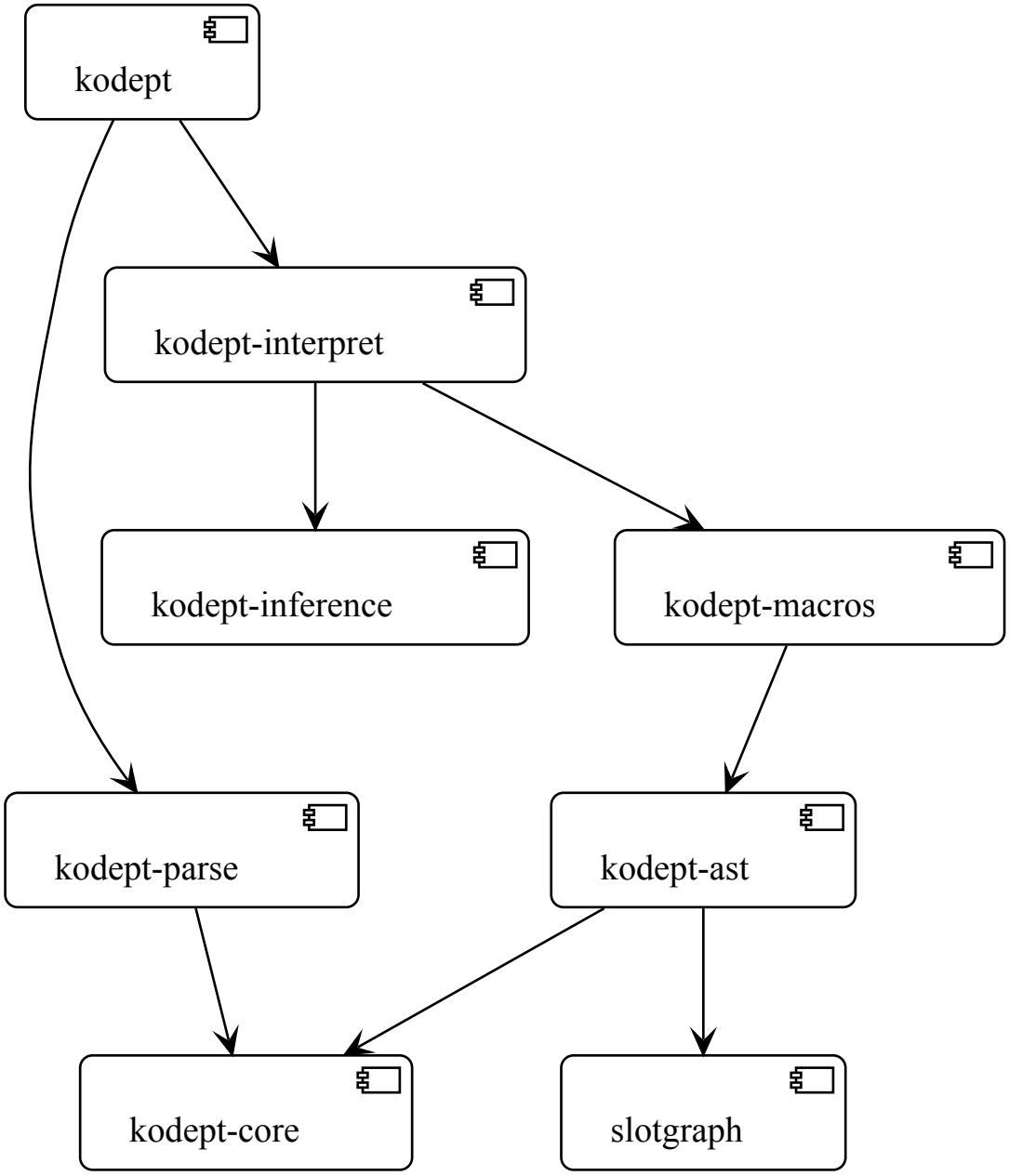

Рисунок 7 – Иерархия модулей в проекте

В рамках этой работы внимание будет сконцентрировано вокруг модулей `kodept-ast`, `kodept-interpret` и `kodept-inference`. Однако, дадим кратное 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.

Дерево разбора - подробная версия AST, куда включается вся информация, полученная от парсеров. Нужно для восстановления конкретной точки в исходном коде при создании диагностики.

В модуле `kodept-ast` определена структура AST и его хранения. Рассмотрим некоторые проблемы, возникшие при проектировании этого модуля.

3.2. Проблема хранения абстрактного синтаксического дерева

Обычно структура AST реализована в программе в виде вложенных друг в друга структур с данными, описывающими тот или иной синтаксис языка (Рисунок 8).

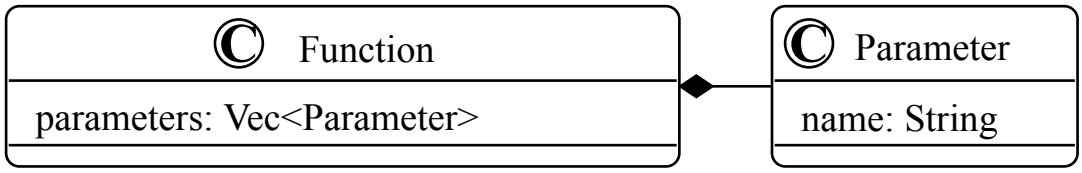

Рисунок 8 – Диаграмма классов, представляющих собой узел функции в AST

Однако при таком подходе возникают определенные трудности. Чтобы написать обход AST, необходимо использовать исключительно рекурсивную реализацию. Это может стать проблемой при формировании большого AST, что его обход приведет к переполнению стека. Кроме того, гораздо большее неудобство возникает и при непосредственно реализации: нужно добавить по отдельной функции обхода для каждого узла дерева. При добавлении нового синтаксиса править код придется сразу в нескольких местах, а это усложняет поддерживаемость.

Поэтому необходимо было придумать более оптимизированный вариант для хранения AST. В итоге было решено переписать имеющиеся структуры так,

чтобы они включали только те данные, которые непосредственно относятся к ним, а также включали цифровой идентификатор (Рисунок 9).

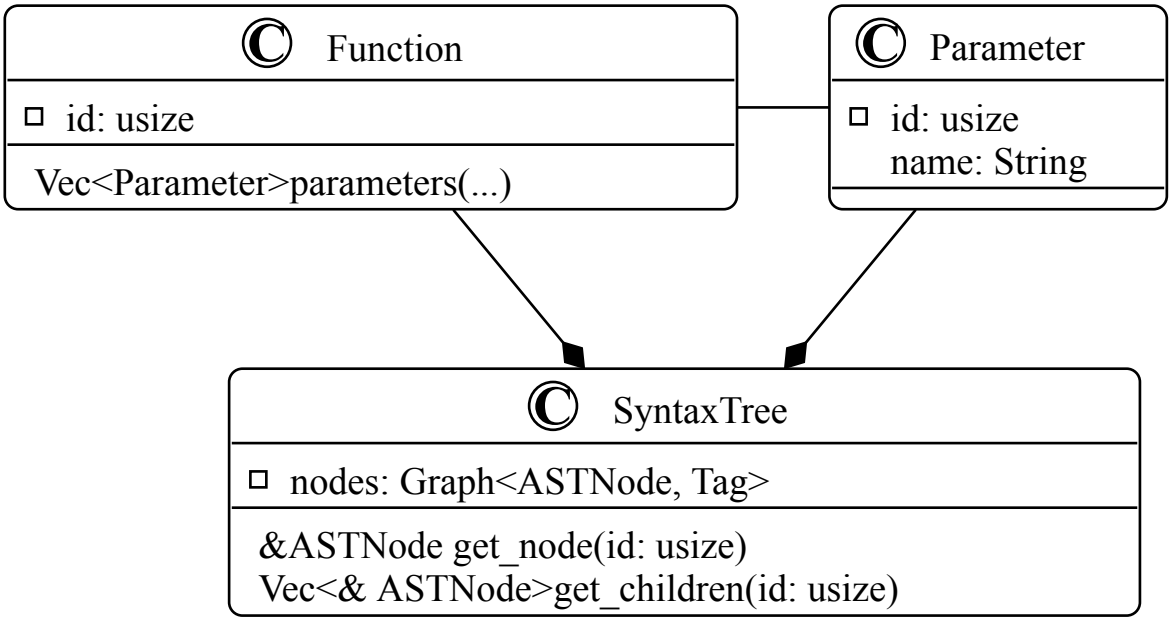

Рисунок 9 – Диаграмма классов после введённого преобразования

Такая композиция позволяет хранить все объекты этих структур в одном месте, линейно. А с помощью индексов моделировать между ними взаимосвязь. Проще говоря, все свелось к хранению обычного графа, где вершины - идентификаторы. С таким видом гораздо удобнее работать, так как алгоритм перебора оперирует числами. Также хранение небольших структур в одном месте повышает вероятность попадания в кеш-память.

При применении «графового» хранения нарушается непосредственная взаимосвязь между структурами. Таким образом, становится непонятно, какие структуры могут быть в каких изначально были «вложены». Но это легко решается с помощью системы типов Rust. А именно: вводятся так называемые свойства принадлежности. Например, функция в качестве вложенных данных могла иметь вектор параметров, возвращаемый тип и тело. Значит, структура Parameter может выступать в качестве «ребенка» структуры Function. Также в структуру Function добавляются методы для доступа к объявленным «детям». Таким образом сохраняется безопасность при работе с AST.

Для наглядного изучения AST, было добавлено сохранение дерева в файл формата DOT [**?**]. В вершинах графа расположены имена синтаксических конструкций и числовой идентификатор. В ребрах - специальный тег, призван-

ный отличать одни вершины от других. На рисунке А.1 изображена визуализация AST программы с листинга ПРИЛОЖЕНИЕ А.1.

3.3. Организация доступа к элементам абстрактного синтаксического дерева

С помощью модуля `kodept-macros` организовывается анализ AST. Во время этого элементы, составляющие дерево, могут быть изменены. К сожалению, из-за специфики Rust, в момент времени может существовать только 1 ссылка на объект, допускающая изменение (exclusive mutable access). Иногда разработчику не обойтись без разделяемого изменяемого состояния. Для этого прибегают к использованию внутренней изменяемости (англ. interior mutability).

Interior mutability позволяет изменять внутренние данные неизменяемой структуры. Самым популярным способом является использование указателя с подсчетом ссылок (reference-counted pointer, RC) вкупе с проверкой заимствований во время выполнения (RefCell). Аналогом из C++ можно считать использование shared_ptr.

Listing 3.1 – Упрощенная реализация структур Rc и RefCell на псевдокоде

```
1  struct Rc<T> {
2      int strong_count;
3      int weak_count;
4      T data;
5  }
6
7  struct RefCell<T> {
8      T data;
9      int borrows_count;
10 }
```

При анализе AST элементы могут поменяться, а структура - нет. Получается, что граф - неизменяемая структура, но его внутренние данные изменяемы. Это как раз подходит под понятие interior mutability. Но использование проверок во время выполнения, а также хранение дополнительных счетчиков для RC плохо влияет на производительность и использование памяти.

Предлагается использовать вместо привычной комбинации из RC + RefCell *zero-cost* абстракцию GhostCell [**?**]. Термином zero-cost или «нулевых

затрат» называют полное отсутствие затрат во время выполнения - все необходимые преобразования выполняет компилятор во время компиляции. Основной идеей GhostCell является разделение ответственности за хранение и модификацию данных. Вводится специальный токен, который дает право изменять данные.

3.4. Реализация алгоритма W

За реализацию алгоритма отвечает модуль `kodept-inference`. В нем также находится определение необходимых термов и типов. Согласно рисунку 7, этот модуль не зависит ни от каких других. Система типов и алгоритм вывода могут быть определены, используя собственную модель. Таким образом, достаточно реализовать функцию по конвертации элементов AST в элементы модели.

3.4.1. Анализ областей видимости

Задачей этого анализа является разбиение AST на области видимости (англ. scopes), при этом строится так называемое дерево областей. Оно состоит из идентификаторов узлов AST, где каждый такой узел начинает новую область. Внутри области можно найти все объявленные переменные и функции, и удостовериться, что не используются необъявленные.

Например, для программы с листинга ПРИЛОЖЕНИЕ А.1, дерево областей будет таким:

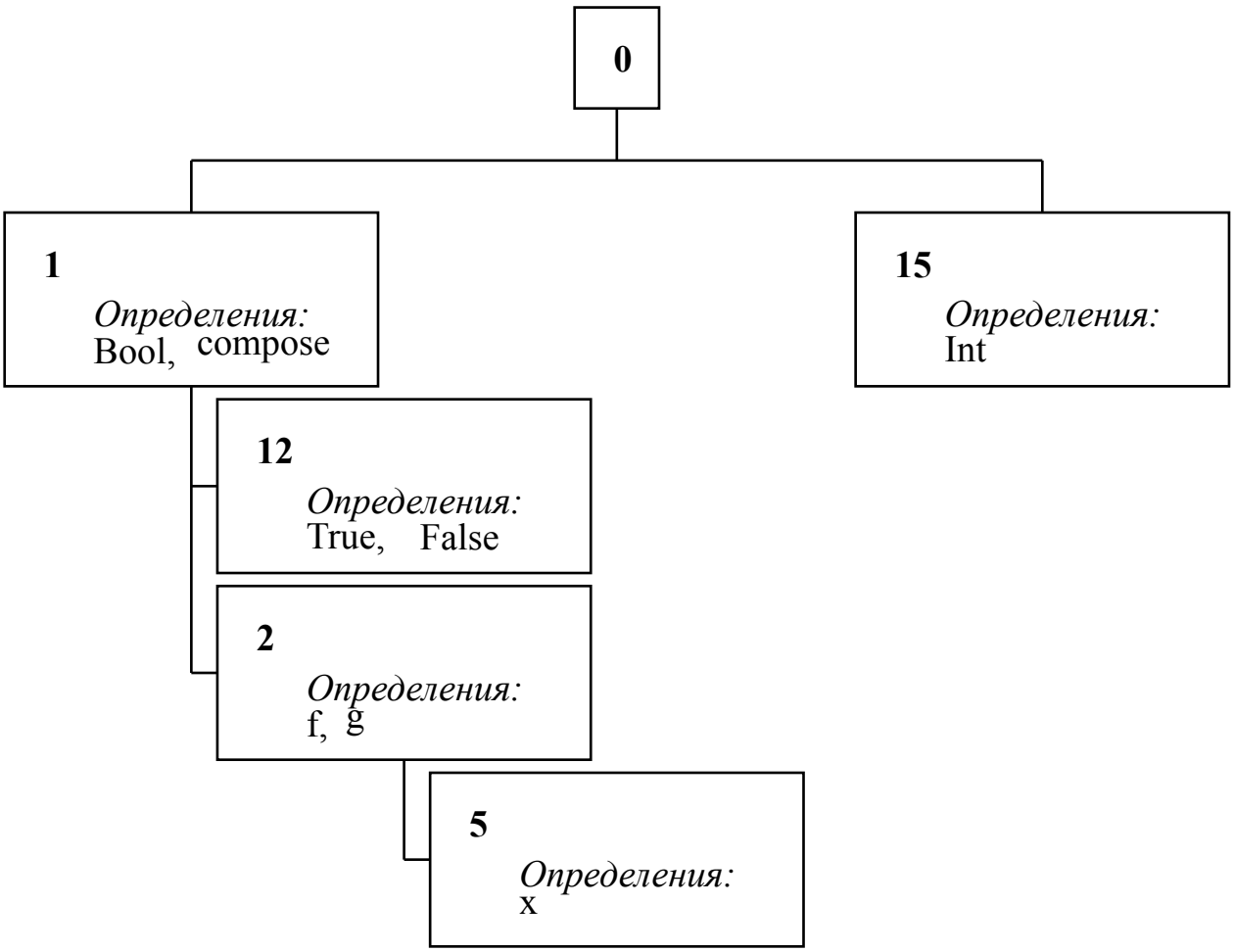

Рисунок 10 – Дерево областей видимости

Для каждого определения из любой области можно применить алгоритм W.

В процессе преобразования модели AST (листинг ПРИЛОЖЕНИЕ Б.1) в термы (2.5.1), также образуются предположения. Они играют роль ограничений. Например, запись `val a: Int = expr` добавит предположение $a : Int$. Таким образом, алгоритм вывода типов проверит, что тип переменной `a` совпадает с типом выражения `expr`.

4. Тестирование и отладка

При разработке приложения не обойтись без тестирования. Оно помогает выявить различные ошибки и исправить их. Популярным вариантом тестирования является модульное тестирование (unit тестирование). Unit test - функция или набор функций, который проверяет корректность работы отдельного нетривиального куска программы.

В ходе разработки компилятора были написаны модульные тесты, они покрывают большое количество кода и успешно выполняются. Для запуска можно использовать систему сборки Cargo. Из папки с проектом следует запустить следующую команду: `cargo test --all-features --all --lib --no-fail-fast`. Cargo соберет проект и последовательно запустит все модульные тесты.

Компилятор Kodept является консольным приложением, поэтому для него был разработан интерфейс командной строки (CLI). С помощью него можно настроить вид выходных данных, поведение работы и др.

На текущий момент поддерживается 3 команды: справка по использованию, генерация графа AST в формате DOT и анализ файла.

Ранее уже были приведены примеры работы команды по генерации графа (А.1) для листинга ПРИЛОЖЕНИЕ А.1. Продемонстрируем работу механизма вывода типов на примере этого же кода. Для этого тоже можно воспользоваться Cargo: `cargo run -- -d examples/test.kd`. В результате в консоль будет выведены тип функции `compose`:

```
cargo run -- -d examples/test.kd
```

`kodept_interpret::type_checker: [compose:` $\forall a, b, c => ('b- >' a) - > ('c- >' b) - >' c - >' a$`]`

Действительно, если преобразовать эту функцию в термы, то получится $\lambda f, g. \lambda x. f(g(x))$. Тип этого выражения действительно совпадает с выведенным типом.

ЗАКЛЮЧЕНИЕ

В результате данной работы был реализован механизм вывода типов для языка программирования Kodept. Показано, что программа действительно может правильно определить тип выражения. Однако развитие проекта на этом не останавливается. Планируется реализация следующего шага в компиляции программы - генератора промежуточного представления.

В ходе работы рассмотрены некоторые проблемы, возникшие при работе с абстрактным синтаксическим деревом. Приведены способы их решения. Успешно решены все поставленные задачи, а именно:

1) сформирована модель абстрактного синтаксического дерева,
2) реализован семантический анализатор, включающий в себя анализатор областей видимости, преобразователь в термы системы типов Хиндли-Милнера и непосредственно механизм вывода типов на основе этой системы.

Литература

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. 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.

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

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

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

6. 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.

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

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

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

10. Barendregt H.P. The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1984. URL: https://books.google.ca/books?id=eMtTAAAAYAAJ.

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

12. 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/.

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

14. GhostCell: separating permissions from data in Rust / Joshua Yanovski, Hoang-Hai Dang, Ralf Jung [и др.] // Proc. ACM Program. Lang. New York, NY, USA, 2021. aug. T. 5, № ICFP. 30 с. URL: https://doi.org/10.1145/3473597.

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)

АКТ
проверки выпускной квалификационной работы

Студент группы *РК6-81Б*

Фамилия Имя Отчество
(Фамилия, имя, отчество)

Тема выпускной квалификационной работы: *[Тема]*

Выпускная квалификационная работа проверена, размещена в ЭБС «Банк ВКР» в полном объеме и соответствует / не соответствует требованиям, изложенным в Положении о порядке
ненужное зачеркнуть
подготовки и защиты ВКР.
Объем заимствования составляет _____ % текста, что с учетом корректного заимствования соответствует / не соответствует требованиям к ВКР _____
ненужное зачеркнуть *бакалавра, специалиста, магистра*

от руки

от руки

подписать

Нормоконтролёр _____ С.В. Грошев
(подпись) (ФИО)

Согласен:
Студент _____ И.О. Фамилия
(подпись) (ФИО)

подписать

Дата: от руки

ПРИЛОЖЕНИЕ А.

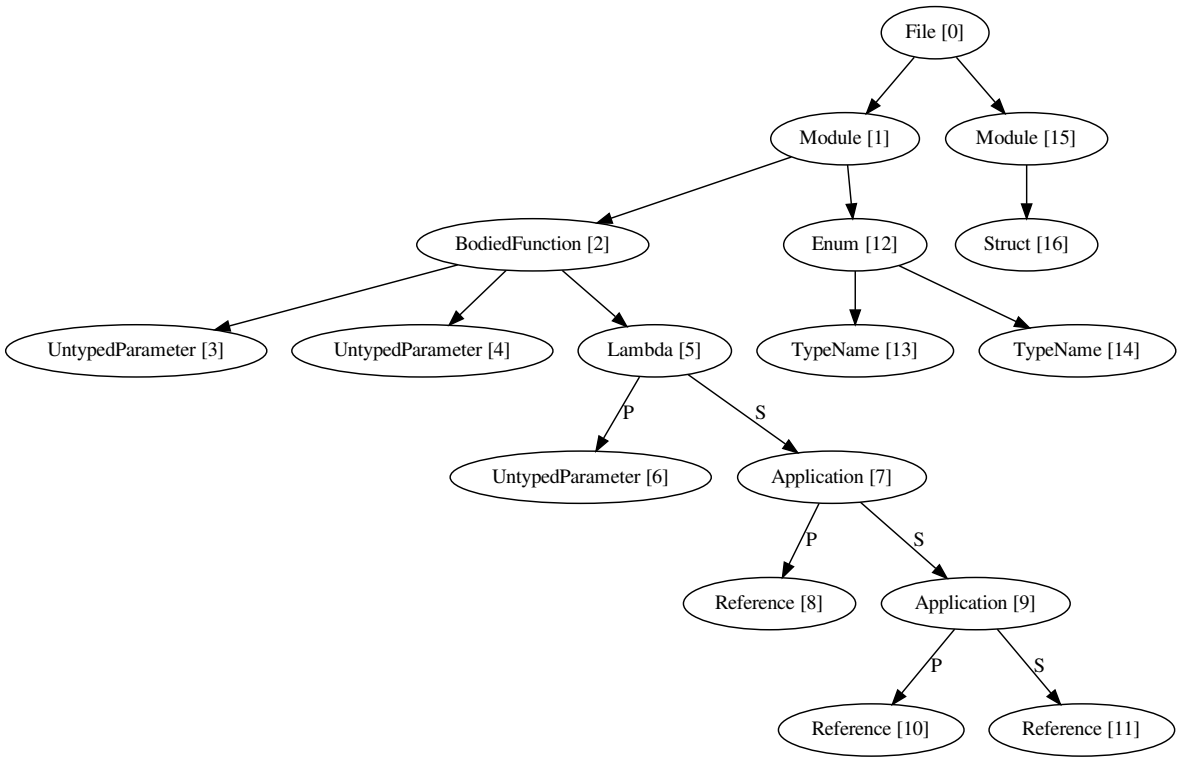

Рисунок А.1 – Изображение структуры абстрактного синтаксического дерева

Listing ПРИЛОЖЕНИЕ 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.

Listing ПРИЛОЖЕНИЕ 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, ...)
```