Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное
учреждение высшего профессионального 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 курсовому проекту

по дисциплине «Модели и методы анализа проектных решений»

на тему

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

Студент РК6-75Б (группа) _____ (подпись, дата) Никитин В.Л. (ФИО)

Руководитель КП _____ (подпись, дата) Соколов А.П. (ФИО)

Консультант _____ (подпись, дата) Соколов А.П. (ФИО)

Москва, 2024

Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное
учреждение высшего профессионального 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 курсового проекта

Студент группы: РК6-75Б

Никитин Владимир Леонидович

(фамилия, имя, отчество)

Тема курсового проекта: Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера

Источник тематики (кафедра, предприятие, НИР): кафедра

Тема курсового проекта утверждена на заседании кафедры «Системы автоматизированного проектирования (РК-6)», Протокол № _____ 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 курсового проекта:

Расчетно-пояснительная записка на 25 листах формата А4.

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

количество: 5 рис., 1 табл., 7 источн.

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

Студент _____ Никитин В.Л.
подпись, дата ФИО

Руководитель курсового проекта _____ Соколов А.П.
подпись, дата ФИО

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

РЕФЕРАТ

курсовой проект: 25 с., 5 рис., 1 табл., 7 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: курсовой проект.

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

Объект исследования: система типов.

Основная задача, на решение которой направлена работа: написание 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

В современном программировании становится все более важным сколько будет потрачено времени на создание того или иного продукта. В это число входит как время, потраченное непосредственно на создание приложения, так и время, потраченное на его поддержку. Поэтому инструменты, применяемые программистом в повседневной работе, должны всячески помочь ему в этом.

Пожалуй, самым главным таким инструментом является компилятор. Разработчики компиляторов прикладывают 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 типы.

В высокоуровневых языках программирования типы окружают разработчика повсюду. Чем более развитая система типов, там 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 спобоб 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, например, какие параметры функции

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. 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 [2], реализующий ее, позволяет также проводить вывод типов.

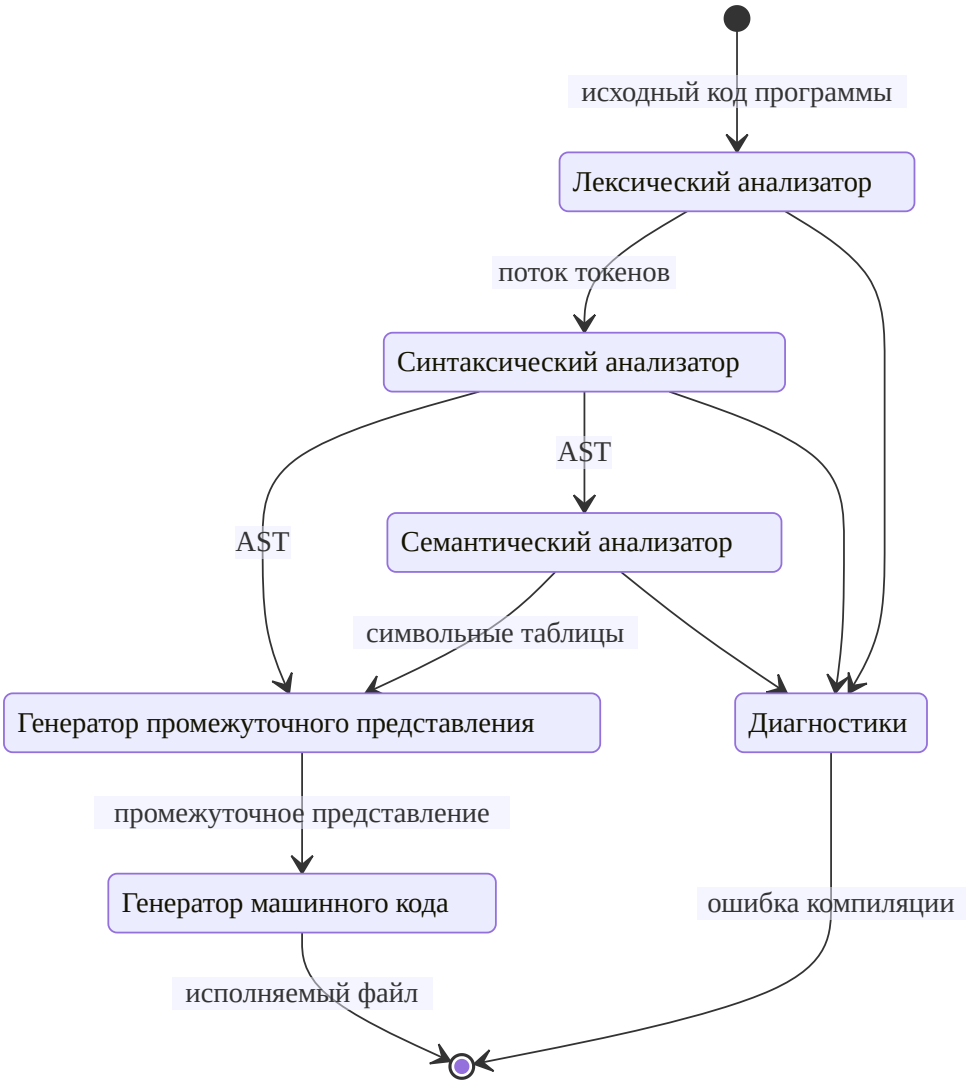

Рисунок В.1. Схема состояний работы компилятора

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

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

1.2.1 Теория типов

В разделе представлена информация о специальном разделе математики - теории типов [3]. Освящены важные понятия - *терм*, *тип*, *суждение* и *система типов*.

Терм x - чаще всего элемент языка программирования, будь то переменная, константа, вызов функции и др. Например, в Haskell, термами будут: лямбда-функция `\x -> x + 1`, определение переменной `let x = "Hello" in ()` и т.д. Как можно заметить, термы могут включать в себя другие термы.

Типом A обозначается метка, приписываемая объектам, 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 же, определяет правила взаимодействия между типами и термами. В программировании это понятие равноценно понятию типизация.

С помощью суждений можно создавать логические конструкции и *правила вывода*. Именно благодаря этому теория типов активно применяется в компиляторах в фазе статического анализа программы, как для вывода, так и для проверки соответствия типов. Более того, согласно изоморфизму Карри-Ховарда [4] (таблица 1), программы могут быть использованы для доказательства логических

высказываний. Такие доказательства называют автоматическими, и они широко применяется среди таких языков, как Agda, Coq, Idris.

Таблица 1. 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: $\exists t : \Gamma \vdash t : T$

Наборы суждений образуют предположения (англ. assumptions), которые образуют контекст Γ. Правила вывода записываются следующим образом, например правило подстановки:

$$\frac{\Gamma \vdash t : T_1, \Delta \vdash T_1 = T_2}{\Gamma, \Delta \vdash t : T_2} \tag{1.1}$$

Выражение 1.1 можно трактовать следующим образом: если в контексте Γ терм t имеет тип T_1, а в контексте Δ тип T_1 равен типу T_2, то можно судить, что при наличии обоих контекстов, терм t имеет тип T_2.

1.2.2 Система типов Хиндли-Милнера

В результате работы над научно-исследовательской работой, было принято решение использовать систему типов Хиндли-Милнера. Среди прочих ее особенностей, важно отметить то, что она способна вывести наиболее общий тип выражения, основываясь на аннотациях типов программиста и окружающем контексте. Предложено использовать её небольшую модификацию с добавлением типов-объединений и некоторых примитивных типов.

Наиболее классическим алгоритмом в этой области является так называемый алгоритм W [2].

Для определения системы типов необходимо 3 составляющие: набор термов, набор типов и набор суждений.

Термы:

$a, b, c ::=$

x	(переменная)
$\lambda x.a$	(лямбда-функция)
$a(b)$	(применение аргумента к функции)
$let\ a = b\ in\ c$	(объявление переменной)
$1, 2, 3, \ldots$	(целочисленный литерал)
$1.1, 1.2, 10.0, \ldots$	(вещественный литерал)
(a, b)	(объединение)

Типы:

$\iota ::=$	(примитивный тип)
$Integral$	(целочистенный)
$Floating$	(вещественный)
$\tau, \sigma ::=$	(мономорфный тип)
ι	
T	(переменная типа)
$\tau \to \sigma$	(функциональный тип)
(τ, σ)	(тип-объединение)
Λ	(пользовательский тип)
$\alpha ::=$	(полиморфный тип)
τ	
$\forall a.\alpha$	(параметрический тип)

Благодаря полиморфным типам имеется возможность определять обобщенные функции. Самый простой пример - функция id. Она имеет следующий тип:

$id : \forall a.a \to a$. Таким образом ее можно вызвать и с аргументом-числом, и с аргументом-функцией.

Работа алгоритма W строится на основе следующих суждений:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{(TAUT)}$$

$$\frac{\Gamma \vdash x : \sigma, \sigma' < \sigma}{\Gamma \vdash x : \sigma'} \quad \text{(INST)}$$

Запись $\sigma' < \sigma$ означает, что тип σ' более конкретный, чем σ.

$$\frac{\Gamma \vdash x : \sigma, a \notin free(\Gamma)}{\Gamma \vdash x : \forall a.\sigma} \quad \text{(GEN)}$$

$$\frac{\Gamma \vdash f : \tau \to \tau', x : \tau}{\Gamma \vdash f(x) : \tau'} \quad \text{(COMB)}$$

$$\frac{\Gamma \cup x : \tau \vdash y : \tau'}{\Gamma \vdash \lambda x.y : \tau \to \tau'} \quad \text{(ABS)}$$

$$\frac{\Gamma \vdash x : \sigma, \Gamma \cup y : \sigma \vdash z : \tau}{\Gamma \vdash (let\ y = x\ in\ z) : \tau} \quad \text{(LET)}$$

$$\frac{\Gamma \vdash x_1 : \tau_1, x_2 : \tau_2, x_3 : \tau_3, \ldots}{\Gamma \vdash (x_1, x_2, x_3, \ldots) : (\tau_1, \tau_2, \tau_3, \ldots)} \quad \text{(TUPLE)}$$

Исходя из этих суждений, алгоритм W составляет так называемое дерево вывода. Если дерево построить удалось, то написанная программа считается верно типизированной.

2 Программная реализация

2.1 Архитектура

В структуре программы почти любого компилятора можно выделить следующие выделяющиеся части:

Рисунок 2. 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 [5]. 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.

Работать с большими проектами в разы удобнее и эффективнее при грамотном разбиении на модули. В экосистеме Rust такие модули именуются крейтами (англ. crates). На диаграмме ниже 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.

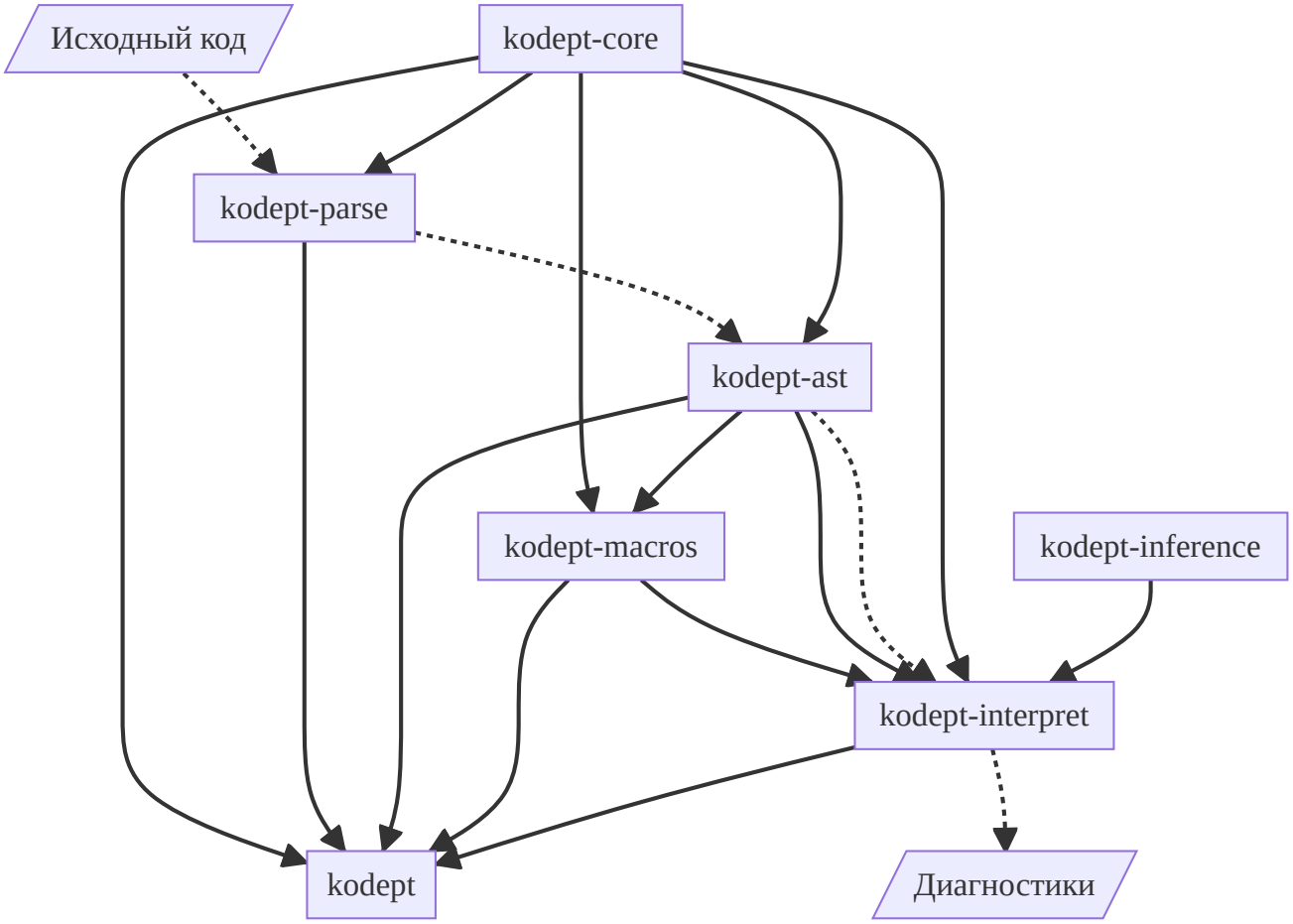

Рисунок 3. Иерархия модулей в проекте

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.

2.2 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 (листинг 2.1).

Листинг 2.1. Псевдокод структур, представляющей собой описание синтаксиса функции в языке программирования.

```
1   struct Function {
2       vector<Parameter> parameters;
3       Type return_type;
4       Statement body;
5   }
6   struct Parameter {
7       string name;
8       Type type;
9   }
10  struct Type {
11      string identifier;
12  }
13  struct Statement {
14      // ...
15  }
```

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.

Листинг 2.2. Псевдокод структур после преобразования.

```
1  struct Function {
2      int id;
3  }
4  struct Parameter {
5      string name;
6      int id;
7  }
8  struct Type {
9      string identifier;
10     int id;
11 }
12 struct Statement {
13     // ...
14     int id;
15 }
```

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.

Листинг 2.3. Упрощенное представление структур после всех доработок на языке Rust

```
struct Function {
    id: usize
}

struct Parameter {
    id: usize,
    name: string
}

impl HasChildrenMarker<Vec<Parameter>> for Function {
    // ...
}

impl Function {
    fn parameters(&self, /* ... */) -> Vec<&Parameter> {
        // ...
    }
}
```

Для наглядного изучения AST, было добавлено сохранение дерева в файл формата DOT [6]. В вершинах графа расположены имена синтаксических конструкций и числовой идентификатор. В ребрах - специальный тег, призванный отличать одни вершины от других. На рисунке А.1 изображена визуализация AST программы с листинга А.1.

2.3 Проблема 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.

Листинг 2.4. 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 [7]. 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.

2.4 Реализация алгоритма W

За реализацию алгоритма отвечает модуль kodept−inference. В нем также находится определение необходимых термов и типов. Согласно рисунку 3, этот мо-

дуль 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.

2.4.1 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 А.1, дерево областей будет таким:

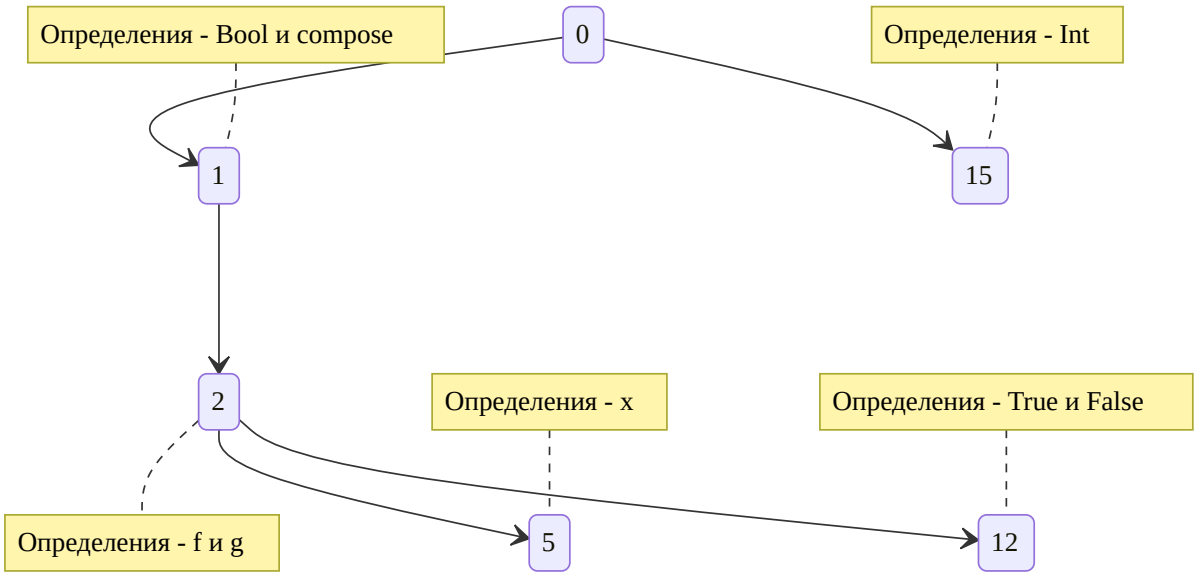

Рисунок 4. Дерево областей видимости

Для каждого определения из любой области можно применить алгоритм W.

В процессе преобразования модели AST (листинг Б.1) в термы (1.2.2), 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`.

3 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 (A.1) для листинга A.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.

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 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 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/.

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

7 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. Т. 5, № ICFP. 30 с. URL: https://doi.org/10.1145/3473597.

Выходные данные

Никитин В.Л.. Разработка механизма вывода типов с использованием системы типов Хиндли-Милнера по дисциплине «Модели и методы анализа проектных решений». [Электронный ресурс] — Москва: 2024. — 25 с. URL: `https://sa2systems.ru:88` *(система контроля версий кафедры РК6)*

Постановка: *доктор физико-математических наук, Соколов А.П.*

Решение и вёрстка: *студент группы РК6-75Б, Никитин В.Л.*

2024, осенний семестр

A

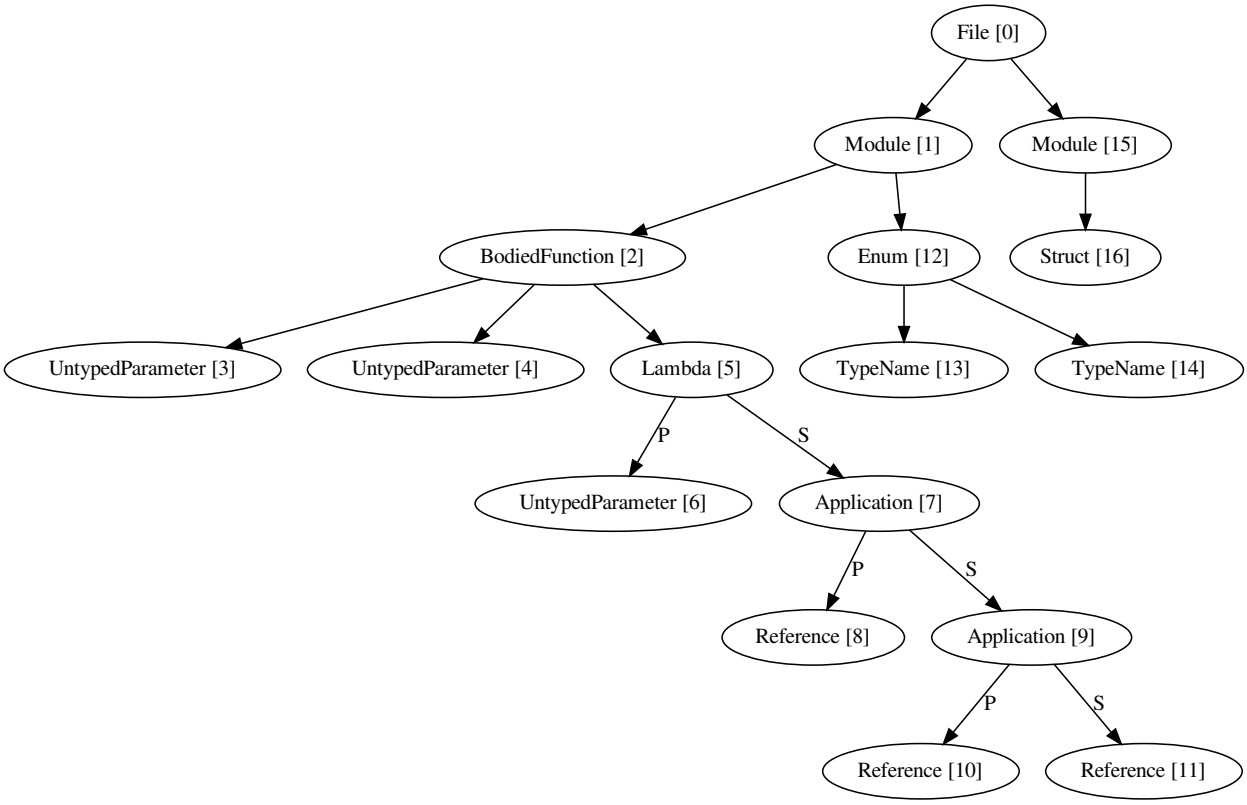

Рисунок А.1. Изображение структуры абстрактного синтаксического дерева

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 }
```

Б

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

```
File, // root element
Module, // module name rest
Struct, // struct name(params) rest
Enum, // enum name rest
TypedParameter, // name: type
UntypedParameter, // name
Variable, // val name: type
InitializedVar, // val name: type = expr
BodiedFunction, // fun name(params) => expr
ExpressionBlock, //  expr1; expr2; ...
Application, // expr(expr)
Lambda, // \binds => expr
Reference, // name
Access, // expr.expr
Number, // number literal
Char, // char literal
String, // string literal
Tuple, // (expr1, expr2, ...)
If, // if expr => expr другие ветки
Elif, // elif expr => expr
Else, // else expr
Binary, // binary operator: +, -, *, /, %, ^
Unary, // unary operator: -, +, !, ~
AbstractFunction, // abstract fun name(params): type
ProdType, // (type1, type2, ...)
```