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 _____ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ_____

КАФЕДРА __ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ_

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Вывод типов пользовательских функций в интерпретаторе формул TeX

Студент ___ИУ9-82б____	_____	__Васянович Д.С.__
(Группа)	(Подпись, дата)	(И.О.Фамилия)
Руководитель ВКР	_____	___Коновалов А.В.___
	(Подпись, дата)	(И.О.Фамилия)
Консультант	_____	_____
	(Подпись, дата)	(И.О.Фамилия)
Консультант	_____	_____
	(Подпись, дата)	(И.О.Фамилия)
Нормоконтролер	_____	_____
	(Подпись, дата)	(И.О.Фамилия)

Москва
2022 г.

АННОТАЦИЯ

Темой денной работы является «Вывод типов пользовательских функций в интерпретаторе формул TeX». Объём данной работы составляет 54 страницы.

Основной объект исследования – определение типов функций, которые задает пользователь, путем введения стадии семантического анализа.

Данная работа состоит из четырёх глав. Первая глава посвящена обзору возможностей реализованного в предыдущих курсовых проектах интерпретатора формул TeX. Во второй главе рассматривается структура системы типов Хиндли-Милнера и ее вариаций. В третьей главе рассматривается реализация алгоритма Хиндли-Милнера в составе стадии семантического анализа, а также приведено руководство по использованию данного интерпретатора. В четвёртой главе описывается процесс тестирования и достигнутые результаты.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной работы – улучшение существующего интерпретатора формул TeX, которое позволит ему выводить типы в функциях, задаваемых пользователем, с помощью системы типов Хиндли-Милнера.

Основная функциональность проекта остается без изменений. На вход интерпретатор получает корректный входной файл, проводит его через стадии анализа (лексический, семантический, и семантический, который и будет реализован в работе). В случае успеха препроцессор вычисляет значения в отмеченных местах и генерирует на выходе корректный файл с результатами вычислений.

Необходимо изучить систему типов Хиндли-Милнера, ее вариации, а также алгоритм вывода типов выражений на ее основе. Далее нужно произвести оптимизацию данного алгоритма для заданного множества языка препроцессора и реализовать стадию семантического анализа, фактически представляющую собой в контексте задачи статическую типизацию.

Также требуется провести тестирование производительности доработки, призванное выявить преимущество статический типизации.

На основе изученного необходимо доработать препроцессор формул TeX.

1 Обзор проекта

1.1 Обзор технологий

TeX является системой компьютерной верстки, созданной в целях создания компьютерной типографии. По сравнению с базовыми текстовыми процессорами, *TeX* позволяет пользователю задавать лишь текст и его структуру, форматируя самостоятельно на основе выбранного пользователем шаблона документ. *TeX* файлы набираются на собственном языке разметки в виде *ASCII*-файлов, содержащих информацию о форматировании текста и/или выводе изображений.

Ядро *TeX* представляет собой язык низкоуровневой разметки, содержащий команды отступа и смены шрифта, имеющий множество готовых наборов макросов и расширений. Наиболее распространённым расширением *TeX* является *LaTeX*.

1.2 Текущая реализация и ее возможности

Данный дипломный проект является расширением курсовых работ Натальи Стрельниковой и Нурсултана Эсенбаева [1].

В первой из них был разработан препроцессор формул *TeX*, автоматически вычисляющий расчет формул в *TeX* документе. В его функционал входили операции с:

1. Арифметическими выражениями
2. Векторами и матрицами, а также арифметические операции с ними (с учетом проверки их размерностей: количества строк и столбцов)
3. Базовыми математическими функциями
4. Циклами
5. Условными операторами
6. Двухмерными графиками функций

7. Пользовательскими функциями, в том числе функциями высшего порядка

В последующей работе была добавлена поддержка следующих возможностей и операций:

1. Оператор суммы
2. Оператор произведения
3. Оператор модуля
4. Операторы округления в большую и меньшую сторону
5. Числа, обладающие размерностью (физические величины) и операции с ними

Логика работы препроцессора построена с учетом использования файла *preproc.tex*, в котором заданы вспомогательные команды, использующиеся для выполнения расчетов в целевом входном файле. Данный файл необходимо подать препроцессору с помощью команды

Листинг 1 – Использование вспомогательного файла с определениями

```
\input {preproc.tex}
```

Расчеты, подлежащее обработке, необходимо включить в окружение *preproc* во вспомогательном файле *preproc.tex*:

Листинг 2 – Использование вспомогательного файла с определениями

```
\newenvironment{preproc}
{\begin{equation*}\begin{array}{1}}
{\end{array}\end{equation*}}
```

Целевые расчет внутри блока *preproc* выполняются с помощью команды *\placeholder*, которая вычисляет значение формулы и помещает его в выходном файле на соответствующем месте, что позволяет препроцессору одновременно вычислять значения и отображать их в финальном документе.

Листинг 3 – Пример расчета выражения

```
\begin{preproc}
    y := f(1, 2)
    y = placeholder[]{} % или y = placeholder{}
\end{preproc}
```

2 Проектирование

2.1 Типизированное лямбда исчисление

В данной работе лямбда исчисление будет использоваться как вспомогательная конструкция для построения системы типов Хиндли-Милнера.

Теория лямбда исчисления представляет собой формальную систему, призванную формализовать понятие вычисления [2].

В основе лежит конструкция базового объекта, то есть терма (λ-термами). Также определены [3]:

1. Аппликация – применение функции *f* к единственному параметру *a*: *f a*. В общем случае *f* может являться не только функцией, но и некоторым алгоритмом.
2. Абстракция – объявление объекта (то есть функции или алгоритма), принимающего на вход единственный параметр *x* и выдающего на выходе другой единственный параметр *t[x],* являющийся результатом применения функции *t* к параметру *x*. Абстракция обозначается как *λ x . t(x).*

Отдельно стоит упомянуть о некоторых дополнительных понятиях, которые входят в конструкцию не типизированного лямбда-исчисления:

1. Альфа-эквивалентность – отношение эквивалентности, подразумевающие идентичность абстракций. Например, функции тождества λ x . x и λ y . y являются альфа-эквивалентными.
2. Бета-редукция – совмещение понятий абстракции и аппликации, то есть процесс аппликации абстракции. Таким образом получается конструкция вида *(λ x . t(x)) a ≡ t[x := a],* применяющая заданную абстракцию *λ x . t(x)* к параметру a.
3. Каррирование – конструкция, позволяющая задавать функции (абстракции) нескольких переменных. Например, функция *f(x, y) = x * y* может быть задана как функция от одной переменной *x* (то есть абстракция), возвращающая функцию от одной переменной *y*:

$\lambda x . \lambda y . x * y$. Аналогичным образом можно построить функцию любого количества переменных, то есть средства лямбда-исчисления позволяют реализовывать многомерные функции.

Рассмотрим теперь случай типизированного лямбда-исчисления. Теперь каждому лямбду-терму соответствует новое синтаксическое понятие – тип. Отсюда вытекают конструкции различных систем типизации. Рассмотрим две основные классификации систем типизаций:

1. Сильная (строгая) и слабая: в случае сильной (строгой) типизации исключаются неявные приведения / преобразования типов средствами компилятора / интерпретатора, в том время как слабая типизации позволяет компилятору / интерпретатору по их собственному «желанию» изменять типы переменных программы в процессе компиляции / интерпретации программы.
2. Статическая и динамическая типизация – в случае статической типизации тип переменной определяется в момент ее объявления и не может быть изменен позднее. В противоположном случае динамической типизации переменная может свободно менять свой тип в контексте программы, в которой она была объявлена.
3. Явная и неявная типизация – в случае явной типизации тип явно объявляется при объявлении переменной, в то время как в противной случае явного указания типа не требуется, а для его определения могут использоваться алгоритмы выведения типа.

В данной работе будет реализовываться неявная статическая строгая типизация.

2.2 Система типов и алгоритм Хиндли-Милнера

Так как интерпретатором формул *TeX* используется концепция неявного объявления типа переменной, то для вывода типов переменных можно использовать систему типов Хиндли-Милнера и алгоритм на ее основе.

Система строится на предположении, что тип любого выражения можно вывести (иногда лишь сделать предположение о типе) путем его вычисления в смысле расчета наиболее общего полиморфного типа.

Процедура вычисления типа выражения опирается на ранее введенные в код программы декларации переменных выражениями с определённым типом, а также на информацию о типах атомарных значений, которая дана «по умолчанию».

Система типов основывается на механизме выводе типов, реализованном в типизированном лямбда-исчислении, который был предложен в 1958 г. Сам алгоритм был окончательно сформулирован в 1969 г. и 1978 г. Роджером Хиндли и Робином Милнером соответственно.

Алгоритм Хиндли-Милнера стремится вывести наиболее общий (полиморфный) тип выражения, используя для этого процесс унификации.

Рассмотрим основные конструкции, использующиеся в типизированном лямбда-исчислении [4]:

1. Базовые компоненты:
    - $\tau ::= \alpha$ – базовый метатип (переменная)
    - $\tau ::= \alpha \rightarrow \alpha'$ - метатип функции (отображения)
2. Вариации лямбда-термов:
    - $e ::= t :: \tau$ – типизированный терм
    - $e ::= x$ – переменная
    - $e ::= t\ t'$ – аппликация t для абстракции t'
    - $e ::= \lambda\ t \rightarrow t'$ – лямбда-абстракция из t в t'

Имеется также набор базовых правил (суждений), исходя из которых рекурсивно выводится тип целевого выражения [5]:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x . t : \tau_1 \to \tau_2} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2} \text{ (T-App)}$$

Рисунок 1 – Базовые правила вывода типов

Где Γ является контекстом выведения типов, то есть набор суждений о типах некоторых перемененных, на основании которого строятся дальнейшие суждения о типах выражений: $\Gamma = \{x_i : \tau_i\}$, i = 1..n.

Знак ⊢ означает корректность типизации выражения справа от него в контексте слева от него, то есть если выражение *e* имеет тип τ в контексте Γ, то оно корректно типизировано в нем: $\Gamma \vdash e : \tau$

Рассмотрим каждое из них подробнее:

1. T-Var – если переменная *x* имеет тип τ в контексте Γ, то она корректно в нем типизирована и имеет в нем тип τ.
2. T-Abs – если в контексте Γ корректно определена переменная *x* типа τ_1 и переменная *t* типа τ_2, то в контексте Γ корректно задана абстракция $\lambda\, x\, .\, t$ типа $\tau_1 \to \tau_2$.
3. T-App – если в контексте Γ корректно типизированы абстракция t_1 типа $\tau_1 \to \tau_2$ и переменная типа t_2 типа τ_1, то в контексте Γ корректно задана аппликация $t_1\ t_2$ типа τ_2.

Таким образом, в результате применения правил типизации получается система уравнений со следующим логичным шагом в виде ее решения с помощью алгоритма унификации.

Данный алгоритм выводит наиболее общий тип для двух выражений, которые уже корректно типизированы какими-то типами. Так как базовый

метатипов всего два, то и правил вывода в процедуре унификации будет четыре (метод не обязательно симметричен) [6]:

1. Переменная – переменная: за общий тип берется тип одной из переменной, а все вхождения другой переменной теперь имеют тип первой переменной.
2. Переменная – отображение: за общий тип берется входной тип отображения и им заменяются типы всех вхождений типовой переменной.
3. Отображение – переменная: если переменная входит в состав отображения, то определить общий тип невозможно и выдается ошибка типизации, в противном случае аналогично предыдущему случаю типы всех вхождений типовой переменной заменяются на результирующий тип отображения.
4. Отображение – отображение: процедура унификации вызывается для входной части первого отображения и результирующей части другого отображения.

2.3 Двухпроходная модификация алгоритма Хиндли-Милнера

В данном проекте за основу будет взята модификация алгоритма, которая, в отличие от базового варианта, производит анализ выражения не только снизу вверх (то есть синтез типа выражения из его термов, тип которых определен), но и сверху вниз (когда выражение должно иметь определённый тип и проверяется на соответствие этому типу).

Суждению $\Gamma \vdash t : \tau$ теперь будет соответствовать пара суждений: утверждение «вывода» $\Gamma \vdash t \Rightarrow \tau$ (то есть в контексте Γ тип выражения t может быть выведен как τ) и утверждение «проверки» (то есть в контексте Γ выражение t должно иметь тип τ). Фактически, получается проверка выражения t с двух сторон, как со стороны прямого вывода типа (то есть его синтезирования), так и

со стороны обратного вывода (то есть проверка выражение на соответствие определенному типу) [7].

Соответствующим образом изменятся и правила вывода типов выражений для двунаправленной модификации алгоритма [8].

Если рассматривать верхнюю часть правила как предпосылку, а нижнюю как заключение, то для каждой составляющей предпосылки и заключения можно определить свойства входа и выхода. Если составляющая нам дана, то она являются известной и входной, в противном случае – свободной и выходной.

Тогда каждое правило будет работать как следующая цепочка действий [9]:

1. Предполагается, что каждая входная составляющая заключения известна.
2. Показывается, что каждая входная составляющая предпосылки известна. При этом каждая выходная составляющая предпосылки по-прежнему является свободной.
3. Предполагается, что каждая выходная составляющая предпосылки является известной.
4. Показывается, что каждая выходная составляющая заключения есть известное выражение.

Входные составляющие помечаются знаком плюс, например, τ^+, в то время как выходная составляющая помечается знаком минус, например, τ^-.

Таким образом, правило T-Var преобразуется в правило BT-Var:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ (\text{BT-VAR})$$

Рисунок 2 – Двунаправленное правило вывода типа для выражения, тип которого известен

То есть, если выражение *x* имеет тип τ в контексте Γ, то в нем можно вывести тип переменной *x* как τ.

Рассмотрим теперь новое по сравнению с классической системой типов Хиндли-Милнера правило, BT-CheckInfer:

$$\frac{\Gamma \vdash t \Rightarrow \tau}{\Gamma \vdash t \Leftarrow \tau}\ (\text{BT-CheckInfer})$$

Рисунок 3 – Двунаправленное правило для случая, когда известна возможность вывода типа выражения

В данном случае, если известно, что в контексте Γ можно вывести тип выражения *t* как τ, то выражение *t* может выдержать проверку на соответствие типу τ.

Для случая типовой аннотация выражения предусмотрено следующее правило, BT-Ann:

$$\frac{\Gamma \vdash t \Leftarrow \tau}{\Gamma \vdash t : \tau \Rightarrow \tau}\ (\text{BT-Ann})$$

Рисунок 4 – Двунаправленное правило для случая, когда дана типовая аннотация к выражению

Таким образом, если дано, что выражение *t* должно иметь тип τ, то тип выражения *t* можно вывести как τ.

При двунаправленной модификации алгоритма необходимо также изменить правило вывода типа абстракции:

$$\frac{\Gamma, x : \tau_1 \vdash t \Leftarrow \tau_2}{\Gamma \vdash \lambda x . t \Leftarrow \tau_1 \to \tau_2}\ (\text{BT-Abs})$$

Рисунок 5 – Двунаправленное правило для случая, когда даны типизированные выражения для конструирования абстракции λ

В случае абстракции меняется направленность типизированных определений. Если в контексте Γ корректно определено выражение x типа τ_1 и выражение t должно иметь тип τ_2, то в контексте Γ абстракция $\lambda x . t$ должна иметь тип $\tau_1 \to \tau_2$ (фактически, получается некая аналогия типовой аннотации, но уже не только для переменной, но и для абстракции).

Наконец, рассмотрим правило аппликации:

$$\frac{\Gamma \vdash t_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 \Leftarrow \tau_1}{\Gamma \vdash t_1\, t_2 \Rightarrow \tau_2}\ (\text{BT-App})$$

Рисунок 6 – Двунаправленное правило для случая, когда даны типизированные выражения для конструирования аппликации $t_1 t_2$

Как и в случае абстракции, меняется направленность типизации выражений, с целью обеспечения известности выходного типа выражения в заключении правила. Так как желателен синтез типа абстракции, то, принимая в контексте Γ то, что тип отображения t_1 может быть выведен как $\tau_1 \to \tau_2$, и проверяя выражение t_2 на соответствие типу τ_1, возможно вывести в контексте Γ тип аппликации $t_1 t_2$ как τ_2 [10].

3 Реализация

3.1 Стадия семантического анализа

В данном дипломной проекте стадия семантического анализа определена для введения статической проверки типов выражений, которая позволяет избежать лишних проверок типов во время стадии времени выполнения и более точно информировать пользователя о ошибках и их местоположении, если такие возникнут.

Так как в текущей реализации проверка типов внутри выражений происходит уже на стадии выполнения, статическая проверка типов (то есть типы выражений будут проверяться ровно один раз, при объявлении этих выражений, что позволит избежать множественных проверок одного и того же куска кода (входного файла) в некоторых случаях, например, в случае цикла) должна обеспечить прирост производительности программы, при прочих равных условиях.

Как и в случае проверки типов во время стадии времени выполнения, в зависимости от разновидности вершины абстрактного синтаксического дерева (далее AST) вызываются взаимоисключающие части кода:

1. В случае корневой вершины (*_tag == ROOT)* идет вызов функции семантического анализа для всех ее дочерних вершин.
2. В случае атомарных значений *(_tag == NUMBER* или *_tag == BEGINM)* происходит возвращения функцией соответствующего типа (с учетом размерностей, так как и число, и элементы матрицы могут быть размерными величинами).
3. В случае вызова значения переменной по идентефикатору (имени переменной) *(_tag == IDENT)* происходит возвращения типа соответствующей этой переменной (если она существует).
4. В случае функции *(_tag == FUNC)* происходит возвращения выходного типа данной функции и проверка аргументов, что передаются для

непосредственного вызова тела функции. В случае, если функции не существует, неверно число аргументов или ошибка в типе хотя бы одного аргумента, программой будет выдана соответствующая ошибка.

5. В случае множественного условного оператора *(_tag == BEGINC)* происходит вызов процедуры проверки для каждой его составляющей, после чего проверяется условие каждой из ветвей по отдельности.
6. В случае оператора унарного плюса, оператора отрицания или левой обычной скобки *(_tag == UADD* или *_tag == NOT* или*_tag == LPAREN)* происходит вызов процедуры проверки для правой части соответствующего вершине выражения.
7. В случае оператора унарного минуса *(_tag == USUB)* происходит вызов процедуры проверки для правой части соответствующего вершине выражения с учетом знака минус текущей вершины.
8. В случае операторов сложения или вычитания, операторов логических операций «и» или «или» и операторов сравнения *(_tag == ADD* или *_tag == SUB* или *_tag == AND* или *_tag == OR* или *_tag == LT* или *_tag == LEQ* или *_tag == GT* или *_tag == GEQ* или *_tag == EQ* (только если правая часть не является элементом расчета выражений: *_tag != PLACEHOLDER*) или *_tag == NEQ)* происходит вызов процедуры проверки для левой и правой частей соответствующего вершине выражения.
9. В случае операторов умножения или деления *(_tag == MUL* или *_tag == DIV* или *_tag == FRAC)* происходит вызов процедуры проверки для левой и правой частей соответствующего вершине выражения.
10. В случае оператора возведения в степень *(_tag == POW)* происходит вызов процедуры проверки для левой и правой частей соответствующего вершине выражения с учетом дополнительных ограничений, накладываемых на данный оператор.
11. В случае операторов суммы или произведения *(_tag == SUM* или *_tag == PRODUCT)* происходит вызов процедуры проверки типа верхней и

нижней границы, а также самого тела оператора. Также, накладываются ограничения на типы и значения верхней и нижней границ суммирования (мультипликации) и учитывается тип оператора при возвращении типа выражения для данной вершины. В случае оператора суммирования достаточно вернуть тип тела, а в случае оператора произведения размерность типа (если она присутствует) необходимо домножить на разницу верхней и нижней границ.

12. В случае размерной величины *(_tag == DIMENSION)* происходит возвращение соответствующего типа с учетом размерности текущей вершины.
13. В случае размерной величины *(_tag == ABS)* происходит вызов процедуры проверки для правой части соответствующего вершине выражения.
14. В случае вершины, в которой происходит подсчет формул и выражений (*_tag == EQ* и у правой части *_tag == PLACEHOLDER*) возвращаемый тип будет соответствовать типу левой части выражения, заключенного в данной вершине.
15. В случае оператора присваивания значения переменной *(_tag == SET)* происходит вызов процедуры проверки для правой части соответствующего вершине выражения. При этом, если у текущей вершины есть вершина-родитель, являющаяся функцией (то есть идет разбор тела функции), то результат выполнения анализа будет учтен для локальной переменной. Если же разбор происходит в глобальном контексте (то есть не внутри какой-либо функции), то в случае идентификатора переменной (не функции) происходит пополнение глобальных переменных, а в случае функции создается новая функция с заданным набором аргументов (то есть тип каждого из аргументов будет определен (по возможности)). Если тип аргумента невозможно вывести, то локальная переменная, ему соответствующая, обладает неопределённым типом и в процессе выполнения может иметь любой

тип, который ей определит пользователь при ее непосредственном вызове из кода входного файла.

16. В случае блока выражения *(_tag == BEGINB)* происходит вызов функции анализа для каждой из его составляющих. Данный случай отличен от случая с корневой вершиной (*_tag == ROOT)*, так как блоки встречаются исключительно внутри функции, следовательно, необходимо учитывать наличие и типы локальных переменных, а также ряд других особенностей анализа выражений внутри пользовательской функции.
17. В случае оператора цикла *(_tag == WHILE)* происходит анализ его условия на предмет нарушений типизации, и, собственно, вызов функции анализа для его тела, результат работы которой и будет выходным типом цикла.
18. В случае расчета формул, а также вызова встроенных функций или констант *(_tag == PLACEHOLDER* или *_tag == KEYWORD* или *_tag == CEIL* или *_tag == FLOOR)* возвращается значение соответствующего типа (почти всегда это будет тип числа).
19. В случае расчета графика, оператора диапазона или списка *(_tag == GRAPHIC* или *_tag == RANGE* или *_tag == LIST)* стадия статического анализа опускается.
20. В случае условного оператора *(_tag == IF)* происходит анализ его условия, а возвращаемый тип будет определяться вызовом функции анализа для правой части вершины, соответствующей данному оператору.
21. В случае оператора транспонирования матрицы *(_tag == TRANSP)* возвращаемый тип будет определяться типом исходной матрицы с учетом транспонирования (для значений матричного типа важна размерность самой таблицы).

3.2 Реализация алгоритма Хиндли-Милнера

В текущей реализации интерпретатор формул *TeX* имеет два основных базовых типа: числовой тип с плавающей точкой (DOUBLE) и матричный тип (MATRIX). Также существует тип функции (FUNC), но в ходе статической проверки типов он фактически не учитывается, так как возникает необходимость точно знать на момент определения тип каждой функции, следовательно, в дальнейшем, при непосредственном вызове в коде, каждой функции будет соответствовать уде базовых тип. Однако, стоит заметить, что тип матрицы MATRIX, по сути, включает в себя числовой тип DOUBLE, так как элементы матрицы имеют тип DOUBLE.

Конструкции абстракции соответствует тип FUNC, а аппликации – вызов функции с подстановкой типовых аргументов (при их наличии).

Остальные конструкции интерпретатора типами непосредственно не являются, однако во время стадии семантического анализа разлагаются на более примитивные составляющие указанных выше типов, которые как раз пригодны для проверки типов [11].

3.2.1 Добавление новых типов

Заметим, что текущих базовых типов недостаточно для реализации алгоритма [12]. Например, в случае, если один из аргументов функции не используется в ее теле:

Листинг 4 – Пример функции, имеющий неиспользуемый аргумент

```
\begin{preproc}
    f(x, y) := x + 5
\end{preproc}
```

Тогда тип аргумента y нельзя определить, но при этом тип функции совершенно логично выводится как DOUBLE.

Потому необходимо введение неопределённого типа (типа неинициализированного выражения): UNDEFINED.

Также, в некоторых случаях тип выражения не может быть выведен напрямую, но может быть сделано предположение о его типе, которое подлежит проверке при непосредственном вызове пользовательской функции. Например:

Листинг 5 – Пример функции, тип значения которой предполагается к выводу

```
\begin{preproc}
    x := 5
    f(y) := x + y
\end{preproc}
```

В таком случае тип функции *f* может быть выведен как DOUBLE, но лишь косвенно. Для таких случаев были введены два дополнительных типа: INFERRED_DOUBLE и INFERRED_MATRIX. Их логика почти что эквивалента типам DOUBLE и MATRIX, однако они лучше помогают отслеживать ошибки в коде программы при нахождении семантическим анализатором ошибок типизации.

3.2.2 Реализация логики синтеза типа выражения

Как было указано ранее, алгоритм анализа является рекурсивным. В зависимости от типа конкретной вершины, находящийся в функции анализа как аргумент, происходит разбор ее составляющих как частей AST.

После этого, когда определены оба типа, и необходимо вывести их наибольший общий тип, в дело вступает процедура унификации. В данной реализации, ввиду отсутствия полиморфизма типов как такого, функции унификации перестала быть рекурсивной и свелась к проверке типов частей выражения.

В случае числе, необходимо убедиться, что оба унифицируемых типа являются либо DOUBLE, либо INFERRED_DOUBLE. Также необходима проверка их размерностей.

Если же одно из выражений имеет тип UNDEFINED, то общий тип может быть выведен как тип другого выражения (если, конечно, оно не UNDEFINED). Например, при унификации выражения типов DOUBLE и UNDEFINED общий тип будет выведен как INFERRED_DOUBLE.

Важно заметить, что два выражения типа UNDEFINED и UNDEFINED нельзя привести к общему типу UNDEFINED. Например:

Листинг 6 – Пример функции, тип аргументов которой не определен

```
\begin{preproc}
    f(x, y) := x + y
    a := f(1, 1 \cdot s)
\end{preproc}
```

Если при анализе функции *f* анализатор не выдаст ошибки типизации, то в случае проверки фактических аргументов функции при присвоении переменной *a* ее значения ошибки тоже не будет, ведь и единица, и одна секунда имеют тип DOUBLE и могут быть подставлены вместо UNDEFINED. Однако их сложение невозможно. Таким образом, операции с UNDEFINED типами являются небезопасными и могут привести к ошибкам во время стадии времени выполнения, потому при попытке унификации двух выражений типа UNDEFINED программа выдает ошибку статической типизации: *Undefined value*.

Также стоит подчеркнуть важное отличие типа DOUBLE от типа INFERRED_DOUBLE. Последний не может использоваться для определения, является ли логическое условие истинным или ложным, так как такой тип не может иметь значения (технически оно имеется как значение по умолчанию, но ввиду специально введенных ограничений не может быть использовано).

3.2.3 Проверка пользовательских функций

Как было сказано выше, проверка функций, заданных пользователем, осуществляется в два этапа.

При объявлении проверяется корректность типизации выходного выражения. При этом подразумевается, что все аргументы функции имеют тип UNDEFINED при старте анализа, как и её локальные переменные.

При этом не допускается тип UNDEFINED как выходное значение функции, так как вследствие этого могут возникнуть ошибки при дальнейшей проверке типов, например:

Листинг 7 – Пример функции, выходной тип которой не определен

```
\begin{preproc}
    f(x) := x
    b := f(1 \cdot s)
    a := b + 1
\end{preproc}
```

Так как выходной тип *f* не может быть определен во время ее объявления, то b также будет иметь тип UNDEFINED (несмотря на то, что по факту в *b* будет лежать тип DOUBLE с размерностью секунд). В случае типизации выражения присваивания значения переменной *a* ошибки тоже не будет, так как единица имеет тип DOUBLE, который вкупе с типом UNDEFINED переменной *b* может быть унифицирован как DOUBLE. Однако во время стадии выполнения возникнет очевидная ошибка при попытке сложения типов DOUBLE, имеющих разную размерность.

Однако в случае аргументов, имеющих тип UNDEFINED на какой-либо момент процесса анализа, ошибка статической типизации выбрасываться не будет.

Например:

Листинг 8 – Пример функции, аргумент которой имеет неопределенный изначально тип (но данный аргумент используется)

```
\begin{preproc}
    f(x) := \begin{block}
        z := x
        y := x + 3
        y + 5
    \end{block} \\
    a := f(1) + 4
\end{preproc}
```

Заметим, что тип переменной *x* не может быть сразу выведен как DOUBLE, так как из выражения *z := x* нельзя сделать никаких выводов о типах выражения и аргумента. При этом, позднее аргумент *x* все же обретает тип INFERRED_DOUBLE, и функция корректно типизируется как INFERRED_DOUBLE, что позволяет ее использовать для сложения с четверкой.

3.3 Оптимизация существующих проверок типов

Так как большая часть операций по проверки типов была переложена со стадии времени выполнения на стадию семантического анализа, то теперь не нужные сравнения типов выражений можно убрать из текущей реализации.

В основном это касается операций проверки размерностей числовых типов DOUBLE и INGERRED_DOUBLE, а также дополнительных проверок внутри матрицы при ее считывании во время непосредственного вычисления выражений:

1. Операторы арифметических операций: сложения, вычитания, умножения и деления, а также возведения в степень.
2. Операторы модуля.
3. Операторы бинарных логических операций.

Листинг 9 – Прототипы функций стадии времени выполнения по анализу арифметических действий

```
static Value plus(const Value &left, const Value &right, const Coordinate& pos);

static Value usub(const Value &arg, const Coordinate& pos);

static Value sub(const Value &left, const Value &right, const Coordinate& pos);

static Value mul(const Value &left, const Value &right, const Coordinate& pos);

static Value div(const Value &left, const Value &right, const Coordinate& pos);

static Value eq(const Value &left, const Value &right, const Coordinate& pos);

static Value le(const Value &left, const Value &right, const Coordinate& pos);

static Value ge(const Value &left, const Value &right, const Coordinate& pos);

static Value lt(const Value &left, const Value &right, const Coordinate& pos);

static Value gt(const Value &left, const Value &right, const Coordinate& pos);

static Value pow(const Value &left, const Value &right, const Coordinate& pos);

static Value abs(const Value &right, const Coordinate& pos);

static Value andd(const Value &left, const Value &right, const Coordinate& pos);

static Value orr(const Value &left, const Value &right, const Coordinate& pos);
```

Также, было необходимо внести типы INFERRED_DOUBLE, INFERRED_MATRIX и UNDEFINED в стадию времени выполнения для

корректного взаимодействия с существующим кодом. Это коснулась как и конструкторов соответствующего класса *Value*, так и операций с объектами данного класса.

3.4 Инструкции по сборке

Для сборки проекта необходим компилятор GCC, версия которого как минимум 9.3.0.

Также требуется система сборки Cmake (как минимум версии 3.5).

Для сборки проекта в UNIX-системах необходимо запустить файл clear_and_build.sh в корневой папке проекта:

Листинг 10 – Запуск файла clear_and_build.sh

```
/bin/bash/ clear_and_build.sh
```

В случае успеха, в корне проекта, в папке cmake-build-debug, будет содержаться исполняемый файл tex-preprocessor, при помощи которого можно будет запустить программу.

3.5 Использование программы

Для запуска программы необходимо выполнить в корневой директории проекта команду:

Листинг 11 – Запуск программы

```
cmake-build-debug/tex-preprocessor
```

Программа выдаст в стандартный поток вывода время своего исполнения. По умолчанию, программа считывает входной файл test.tex и записывает результат работы в файл _test.tex. Это поведение можно

изменить, подав на вход другой файл и определив выход как другой файл. Для этого нужно изменить строки 7 и 8 в файле CMakeLists.txt. задав нужные относительные пути к входному и выходному файлам соответственно, и пересобрать проект с помощью скрипта clear_and_build.sh, как было показано выше.

Для запуска тестирования производительности можно запустить скрипт run_tests.sh:

Листинг 12 – Пример изменения скрипта run_tests.sh

```
/bin/bash/ run_tests.sh
```

После этого, по результатам запуске скрипта run_tests.sh, в стандартный поток вывода будет выведено четыре числа, например:

Листинг 13 – Пример выходных данных скрипта run_tests.sh

```
85.82779655172413793103
85.2111
85.5401
86.0694
```

Что означает, соответственно:

- среднее время работы программы
- нижний (первый) квартиль времени работы программы (25% перцентиль)
- медиана (второй квартиль) времени выполнения
- верхний (третий) квартиль времени работы программы (75% перцентиль)

за период тестирования (по умолчанию производится 29 тестовых запусков программы подряд).

4 Тестирование

4.1 Тестирование работоспособности интерпретатора

Так как данный проект является доработкой существующей программы, необходимо убедиться, что существующий функционал остался работоспособен. Проверим несколько операций с различными (в том числе размерными) величинами:

1. Арифметические операции:

Листинг 14 – Примеры корректных арифметических операций

```
\begin{preproc}
    x := 1 \cdot s \cdot kg + 2.5 \cdot kg \cdot s
    x := 2 \cdot s * x / 34
    x = \placeholder{}

    y := 24 \cdot s \cdot kg
    y = \placeholder{}

    z := y > x
    z = \placeholder{}
\end{preproc}
```

Заметим, что переменная типа *x* имеет итоговую размерность *кг * $с^2$*, что позволяет ее сравнивать с перемененной *y* эквивалентного типа (порядок величин-размерностей не важен). Корректность операции сравнения также истинна ввиду равенства типов *y* и *x*.

Листинг 15 – Примеры некорректных арифметических операций

```
\begin{preproc}
    x := 1 \cdot s \cdot kg + 2.5 \cdot kg \cdot s \cdot s
    x = \placeholder{}

    y := 24 \cdot s \cdot kg
    y = \placeholder{}

    z := -35 \cdot kg
    z = \placeholder{}

    a := y < z
    a := \placeholder{}
\end{preproc}
```

На листинге выше, для первого выражения, программа выдаст логичную ошибку несоответствия типов в операции сложения. В случае анализа переменной *a* возникла бы схожая ошибка: сравнение чисел с различными размерностями невозможно.

Листинг 16 – Примеры ошибок во время анализа некорректных арифметических операций

```
Cannot ADD/SUB/AND/OR/LT/LEQ/GT/GEQ/EQ/NEQ non double (or with
different dimension) value: 1 \cdot kg \cdot s and value: 1 \cdot
kg \cdot s^2 in node: ADD

Cannot ADD/SUB/AND/OR/LT/LEQ/GT/GEQ/EQ/NEQ non double (or with
different dimension) value: 1 \cdot kg \cdot s and value: 1 \cdot
kg in node: LT
```

2. Пользовательские функции: особенности статической типизации

   Приведем несколько примеров функций, характеризующих поведение семантического анализатора:

Листинг 17 – Примеры пользовательских функций

```
\begin{preproc}

        test1() := 2 \\
        test1() = \placeholder{} \\

        x := 2
        y := 3
        test2() := x + y
        test2() = \placeholder{} \\

        test3(a) := a * 2
        test3(7) = \placeholder{} \\

        test4(z) := y * z + x * z
        test4(7) = \placeholder{} \\

        test5(a, b) := test3(a) + test4(b)
        test5(4, 5) = \placeholder{} \\
        test5(test3(x), test4(y)) = \placeholder{} \\

        switch1(c) := \begin{caseblock}
                        1 \cdot kg \when x > 0 \\
                        1 \cdot m \when x < 0 \\
                        1 \otherwise
        \end{caseblock}

%       ------------------------------------------------

        test5(c, d) := c(x) + y * d

        test6(c, d) := c * d

        x := 1 \cdot s
        y := 2 \cdot kg

        test7(z) := x * y + y * x + z
        test7(1 \cdot s) = \placeholder{} \\

        switch2(c) := \begin{caseblock}
                        1 \cdot kg \when c > 0 \\
                        1 \cdot m \when c < 0 \\
                        1 \otherwise
        \end{caseblock}
\end{preproc}
```

- Функция *test1* не имеет аргументов и возвращает корректное числовое значение, потому для нее проверка типов тривиальна.
- Далее, введя глобальные переменные *x* и *y*, можно определить функцию *test2*, которая будет возвращать значение типа DOUBLE, выведенное из типов переменных *x* и *y*.
- Функция *test3* будет выдавать тип INFERRED_DOUBLE, выведенный из типа ее единственного аргумента a, использующегося в операции умножения с двойкой, которая, очевидно, имеет тип DOUBLE.
- Функция *test4* выведет тип своего аргумента *z* как INFERRED_DOUBLE, исходя из глобальных переменных *x* и *y* и операции сложении данный переменных с UNDEFINED на момент объявления *z*, которая в результате процедуры унификации вернет тип INFERRED_DOUBLE.
- Функция *test5* является суммой двух предыдущее объявленных функций: *test3* и *test4*, корректно типизированных как INFERRED_DOUBLE, потому итоговый тип будет выведен как INFERRED_DOUBLE. Заметим, что в данном случае аргументы *a* и *b* функции *test5* обязаны будут иметь тип INFERRED_DOUBLE, так как аргументы соответствующих функций *test3* и *test4* имеют аргументы именно такого типа.
- Функция *switch1* имеет блок множественного условного оператора, содержащий условия сравнения переменной с числом. Но так как глобальная переменная *x*, участвующая в сравнении, имеет корректно определенный тип DOUBLE, то тип конструкции *caseblock* можно вывести как DOUBLE с размерностью *кг*, потому как *x* имеет значение 2, которое больше 0, потому, независимо от типа аргумента *c* функции *switch1*, во время вызова будет выполняться лишь данная ветвь оператора и возвращаемый тип

функции (DOUBLE с размерностью *кг*) не будет зависеть от ее аргумента, имеющего тип UNDEFINED.

- Функция *test6* имеет некорректно типизированную левую часть оператора сложения. Несмотря на то, что правая часть выводится как INFERRED_DOUBLE благодаря глобальной переменной *y*, имеющий тип DOUBLE, левая часть содержит вызов неизвестной функции *c*, которая является аргументом функции *test6*. Несмотря на то, что унификация INFERRED_DOUBLE и UNDEFINED возможна, вызов функции c(x) не является выражением идентификатора, а, следовательно, к нему не может быть применена процедура унификации типов. Это сделано для того, чтобы не пропускать наружу случаи, в которых тип возвращаемого значения функции *c* определялся бы не как DOUBLE соответствующей размерности. Потому объявление функции *test6* вернет ошибку типизации:

Листинг 18 – Ошибка типизации функции *test6*

```
Undefined value: undefined in node: ADD
```

- Функция *test7* имеет ошибку типизации умножения, так как оба ее аргумента имеют тип UNDEFINED. Потому выходной тип данной функции не может быть выведен как UNDEFINED по причинам, изложенным выше.

Листинг 19 – Ошибка типизации функции *test7*

```
Undefined value: undefined in node: MUL
```

- Переопределив глобальные переменные *x* и *y* как числа с размерностями *с* и *кг*, введена функция *test8*. Ее тело не содержит ошибок, так как процедура типизации корректно определяет аргумент *z* как выражение типа INFERRED_DOUBLE. Однако ее

вызов с аргументом типа DOUBLE, но с другой размерностью (*с* против *кг* * *с*), приводит к ошибке типизации по причине несоответствия размерностей:

Листинг 20 – Ошибка типизации функции *test8*

```
FUNC argument has an incorrect type (or different dimensions):
DOUBLE instead of: INFERRED_DOUBLE in node: test8
```

- Функция *switch2* содержит в своем теле блок множественного условного оператора, в котором участвует сравнение с числом. Но на этот раз первые две ветви альтернативы содержат уже сравнение типа DOUBLE (число 0) и типа UNDEFINED (выражение *с*). Процедура унификации в данном случае неприменима, так как операторы сравнения запрещают унификацию со значениями типа UNDEFINED во избежание ошибок типизации во время непосредственного вызова функции. Поэтому интерпретатор выдает ошибку:

Листинг 21 – Ошибка типизации функции *switch2*

```
Cannot compare using inferred double value: 0.0 in node: ALT
```

3. Проверка существующего функционала: различные конструкции
    - Пример программы с операциями с размерностями мощности и энергии. На листингах 22 и 23 ниже приведены входные и выходные файлы соответственно:

Листинг 22 – Тестовый расчет мощности: на вход

```
\begin{preproc}
  N := kg \cdot m \cdot s^{-2}
  J := N \cdot m
  W := J / s
  hour := 3600 \cdot s
  kWh := 1000 \cdot W \cdot hour
  cal := 4.1868 \cdot J
  kcal := 1000 \cdot cal
  kWh = \placeholder[kcal]{}
\end{preproc}
```

Листинг 23 – Тестовый расчет мощности: на выход

```
\begin{preproc}
  N := kg \cdot m \cdot s^{-2}
  J := N \cdot m
  W := J / s
  hour := 3600 \ cdot s
  kWh := 1000 \cdot W \cdot hour
  cal := 4.1868 \ cdot J
  kcal := 1000 \ cdot cal
  kWh = \ placeholder [ kcal ]{859.84523}
\end{preproc}
```

- Пример программы с пользовательской функцией округления в меньшую сторону и графиком к ней. На листингах 24 и 25 ниже приведены входные и выходные файлы соответственно:

Листинг 24 – Функция округления в меньшую строну и ее график: на вход

```
\begin{preproc}
ceil (x) := \begin{block}
  x := x \cdot \pi - \pi / 2 \\
  frac := \arctan(\tan(x)) \\
  (x - frac) / \pi
\end{block} \\

ceil (0.4) = \placeholder{} \\
ceil (0.5) = \placeholder{} \\
ceil (0.6) = \placeholder{} \\
ceil (1.0) = \placeholder{} \\
ceil (2.1) = \placeholder{} \\

\graphic{ceil}{\range[0.1]{0}{4.9}}
  {(0.000000 ,0.000000)}
\end{preproc}
```

Листинг 25 – Функция округления в меньшую строну и ее график: на выход

```
\begin{preproc}
ceil (x) := \begin{block}
  x := x \cdot \pi - \pi / 2 \\
  frac := \arctan(\tan(x)) \\
  (x - frac) / \pi
\end{block} \\

ceil (0.4) = \placeholder{0} \\
ceil (0.5) = \placeholder{0} \\
ceil (0.6) = \placeholder{0} \\
ceil (1.0) = \placeholder{1} \\
ceil (2.1) = \placeholder{2} \\

\graphic{ceil}{\range[0.1]{0}{4.9}}
{(0.000000 ,0.000000)
(0.100000  ,0.000000)  (0.200000  ,0.000000)  (0.300000  ,0.000000)
(0.400000  ,0.000000)  (0.500000  ,0.000000)  (0.600000  ,0.000000)
(0.700000  ,0.000000)  (0.800000  ,0.000000)  (0.900000  ,0.000000)
(1.000000  ,0.000000)  (1.100000  ,1.000000)  (1.200000  ,1.000000)
(1.300000  ,1.000000)  (1.400000  ,1.000000)  (1.500000  ,1.000000)
(1.600000  ,1.000000)  (1.700000  ,1.000000)  (1.800000  ,1.000000)
(1.900000  ,1.000000)  (2.000000  ,2.000000)  (2.100000  ,2.000000)
(2.200000  ,2.000000)  (2.300000  ,2.000000)  (2.400000  ,2.000000)
(2.500000  ,2.000000)  (2.600000  ,2.000000)  (2.700000  ,2.000000)
(2.800000  ,2.000000)  (2.900000  ,2.000000)  (3.000000  ,3.000000)
(3.100000  ,3.000000)  (3.200000  ,3.000000)  (3.300000  ,3.000000)
(3.400000  ,3.000000)  (3.500000  ,3.000000)  (3.600000  ,3.000000)
(3.700000  ,3.000000)  (3.800000  ,3.000000)  (3.900000  ,3.000000)
(4.000000  ,4.000000)  (4.100000  ,4.000000)  (4.200000  ,4.000000)
(4.300000  ,4.000000)  (4.400000  ,4.000000)  (4.500000  ,4.000000)
(4.600000  ,4.000000)  (4.700000  ,4.000000)  (4.800000  ,4.000000)
(4.900000 ,4.000000)
}\end{preproc}
```

4.2 Измерение производительности интерпретатора

Так как основная цель введения статической проверки типов заключалась в оптимизации ненужных вычислений (то есть проверок типов), то производительность уместно измерять в контексте скорости работы предыдущей версии программы, написанной без использования стадии семантического анализа вкупе с проверкой типов выражений.

Для корректного сравнения для каждого тестового случая и для обеих версий программ будут посчитаны нижний и верхний квартили (25 и 75 перцентили соответственно), образующие интерквантильный размах и 50 доверительный интервал. Тогда медиана (50% перцентиль) определяет типичное значение. Были проведены 29 тестовых прогонов для кода «до» и для кода «после», вычислены нижний и верхний квартили, подсчитаны медианное и среднее значения. Для всех тестов значения приведены в миллисекундах (мс).

1. Рассмотрим сначала входной файл небольшого размера (в контексте количества операций, исполняемых на стадии времени выполнения) без размерных величин. Входные данные приведены на листинге 26, а результаты тестирования – в таблице 1.

Листинг 26 – Файл малого размер, без размерных величин

```
\usepackage{amsmath}
\begin{preproc}
x := 1
x := 100
z := 228

f(x, y) := x * (x + y)
v := \begin{pmatrix}
1 \\ 2 \\ z
\end{pmatrix}
v = \placeholder {}\\
\graphic{f}{\range{0}{1}, 0.5}{}\\
\end{preproc}
```

Таблица 1 – Производительность: файл малого размера, без размерных величин

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	0.342727	0.365751
Медиана	0.382338	0.429244
Среднее значение	0.417817	0.541862
Верхний квартиль	0.454681	0.576020
Доверительный интервал 50%	0.342727-0.454681 (0.111954)	0.365751-0.576020 (0.210269)

Медианный прирост в данном случае составил 10.93%, в то время как среднее время работы уменьшилось на 22.90%. Заметим, что в данном случае нельзя говорить о содержательном выигрыше в скорости, так как доверительные интервалы результатов тестирования пересекаются.

2. Рассмотрим далее входной файл небольшого размера (в контексте количества операций, исполняемых на стадии времени выполнения) с размерными величинами. Входные данные приведены на листинге 27, а результаты тестирования – в таблице 2.

Листинг 27 – Файл малого размера, с размерными величинами

```
\usepackage{amsmath}
\begin{preproc}
x := 1 \cdot s \cdot kg
x := 100 \cdot s \cdot kg
z := 228 \cdot s \cdot kg

f(x, y) := x + y
v := \begin{pmatrix}
1 \cdot s \cdot kg \\ 2 \cdot s \cdot kg \\ z
\end{pmatrix}
v = \placeholder {}\\
\graphic{f}{\range{0}{1}, 0.5}{}\\
\end{preproc}
```

Таблица 2 – Производительность: файл малого размера, с размерными величинами

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	0.413732	0.424932
Медиана	0.454963	0.475989
Среднее значение	0.546994	0.551372
Верхний квартиль	0.614975	0.683985
Доверительный интервал 50%	0.413732-0.614975 (0.210156)	0.424932-0.683985 (0.259053)

Медианный прирост в данном случае составил 4.42%, в то время как среднее время работы уменьшилось на 0.79%. Заметим, что в данном случае нельзя говорить о содержательном выигрыше в скорости, так как доверительные интервалы результатов тестирования пересекаются.

3. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) без размерных величин. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Входные данные приведены на листинге 28, а результаты тестирования – в таблице 3.

Листинг 28 – Файл большого размера с циклом, с размерными величинами

```
\begin{preproc}
    i := 0 \cdot s
    step := 1 \cdot s
    limit := 100000 \cdot s

    x := \while{i < limit}
            \begin{block}
                i := i + step
            \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 3 – Производительность: файл большого размера с циклом, с размерными величинами

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	85.2111	101.519
Медиана	85.5401	101.998
Среднее значение	85.8277	102.172
Верхний квартиль	86.0694	102.687

Доверительный интервал 50%	85.2111-86.0694 (0.8583)	101.519-102.687 (1.168)

Медианный прирост в данном случае составил 16.14%, в то время как среднее время работы уменьшилось на 16.00%. Заметим, что в данном случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

4. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) без размерных величин. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Входные данные приведены на листинге 29, а результаты тестирования – в таблице 4.

Листинг 29 – Файл большого размера с циклом, без размерных величин

```
\begin{preproc}
    i := 0
    step := 1
    limit := 100000

    x := \while{i < limit}
             \begin{block}
                 i := i + step
             \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 4 – Производительность: файл малого размера с циклом, без размерных величин

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	84.9486	101.181

Медиана	85.1191	101.369
Среднее значение	85.3998	101.596
Верхний квартиль	85.8901	102.225
Доверительный интервал 50%	84.9486-85.8901 (0.9415)	101.181-102.225 (1.044)

Медианный прирост в данном случае составил 16.03%, в то время как среднее время работы уменьшилось на 15.98%. Заметим, что в данном случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

5. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) с размерными величинами. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Также в цикле вызывается функция, содержащая размерные величины. Это сделано с целью проверки производительности анализа пользовательских функций, содержащих размерные величины. Входные данные приведены на листинге 30, а результаты тестирования – в таблице 5.

Листинг 30 – Файл малого размера с циклом, с размерными величинами и пользовательскими функциями

```
\begin{preproc}
    f(x) := 1 \cdot s + 2 \cdot s * x - 4 \cdot s / x - 2 \cdot s
* x + 4 \cdot s / x

    i := 0 \cdot s
    step := 1
    limit := 100000 \cdot s

    x := \while{i < limit}
    \begin{block}
        i := i + f(step)
    \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 5 – Производительность: файл большого размера с циклом, с размерными величинами и пользовательскими функциями

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	783.734	841.707
Медиана	784.811	849.694
Среднее значение	784.586	863.372
Верхний квартиль	786.706	868.664
Доверительный интервал 50%	783.734-786.706 (2.972)	841.707-868.664 (26.957)

Медианный прирост в данном случае составил 7.63%, в то время как среднее время работы уменьшилось на 9.13%. Заметим, что в данном

случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

6. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) без размерных величин. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Также в цикле вызывается функция. Это сделано с целью проверки производительности анализа пользовательских функций. Входные данные приведены на листинге 31, а результаты тестирования – в таблице 5.

Листинг 31 – Файл малого размера с циклом, без размерных величин и с пользовательскими функциями

```
\begin{preproc}
    f(x) := 1 + 2 * x - 4 / x - 2 * x + 4 / x

    i := 0
    step := 1
    limit := 100000

    x := \while{i < limit}
    \begin{block}
        i := i + step * f(step)
    \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 6 – Производительность: файл большого размера с циклом, без размерных величин и с пользовательскими функциями

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	572.934	631.713

Медиана	574.446	647.605
Среднее значение	579.484	662.686
Верхний квартиль	576.396	688.718
Доверительный интервал 50%	572.934-576.396 (3.462)	631.713-688.718 (57.005)

Медианный прирост в данном случае составил 11.30%, в то время как среднее время работы уменьшилось на 12.56%. Заметим, что в данном случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

7. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) с размерными величинами. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Также в цикле вызывается функция, содержащая другие функции и размерные величины. Это сделано с целью проверки производительности анализа пользовательских функций, использующих другие функции в своем теле. Входные данные приведены на листинге 32, а результаты тестирования – в таблице 7.

Листинг 32 – Файл большого размера с циклом, с размерными величинами и пользовательскими функциями, содержащими другие функции

```
\begin{preproc}
    f1(x) := 1 \cdot s + 2 \cdot s * x - 4 \cdot s / x - 2 \cdot s
* x + 4 \cdot s / x

    f2(x) := 1 \cdot s + 2 \cdot s * x - 4 \cdot s / x - 2 \cdot s
* x + 4 \cdot s / x + f1(x)

    f3(x) := 1 \cdot s + 2 \cdot s * x - 4 \cdot s / x - 2 \cdot s
* x + 4 \cdot s / x + f2(x)

    i := 0 \cdot s
    step := 1
    limit := 100000 \cdot s

    x := \while{i < limit}
    \begin{block}
        i := i + f3(step)
    \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 7 – Производительность: файл большого размера с циклом, с размерными величинами и пользовательскими функциями, содержащими другие функции

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	1249.50	1292.72
Медиана	1253.25	1296.72
Среднее значение	1253.29	1296.40

Верхний квартиль	1258.88	1301.92
Доверительный интервал 50%	1249.50-1258.88 (9.38)	1292.72-1301.92 (9.20)

Медианный прирост в данном случае составил 3.35%, в то время как среднее время работы уменьшилось на 3.33%. Заметим, что в данном случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

8. Рассмотрим далее входной файл большого размера (в контексте количества операций, исполняемых на стадии времени выполнения) без размерных величин. Большое количество операций на стадии времени выполнения достигается за счет использования цикла. Также в цикле вызывается функция, содержащая другие функции. Это сделано с целью проверки производительности анализа пользовательских функций, использующих другие функции в своем теле. Входные данные приведены на листинге 33, а результаты тестирования – в таблице 8.

Листинг 33 – Файл большого размера с циклом, с размерными величинами и пользовательскими функциями, содержащими другие функции

```
\begin{preproc}
    f1(x) := 1 + 2 * x - 4 / x - 2 * x + 4 / x

    f2(x) := 1 + 2 * x - 4 / x - 2 * x + 4 / x + f1(x)

    f3(x) := 1 + 2 * x - 4 / x - 2 * x + 4 / x + f2(x)

    i := 0
    step := 1
    limit := 100000

    x := \while{i < limit}
    \begin{block}
        i := i + f3(step)
    \end{block}

    x = \placeholder{}
\end{preproc}
```

Таблица 8 – Производительность: файл большого размера с циклом, с размерными величинами и пользовательскими функциями, содержащими другие функции

	Версия со статической проверкой типов	Версия без статической проверки типов
Нижний квартиль	875.944	909.690
Медиана	877.339	914.812
Среднее значение	878.840	926.599
Верхний квартиль	880.111	934.708

Доверительный интервал 50%	875.944-880.111 (4.167)	909.690-934.708 (25.018)

Медианный прирост в данном случае составил 4.10%, в то время как среднее время работы уменьшилось на 5.15%. Заметим, что в данном случае имеется содержательный выигрыш в скорости, так как доверительные интервалы результатов тестирования не пересекаются.

Во всех случаях код категории «до» брался по коммиту с хэшем $9127d3e$..., а код категории «после» имел хэш e2ccedc.... Тестирование проводилось с использованием компилятора g++ версии 9.3.0 с флагом $-O3$ и системой сборки Cmake версии 3.5 на ПК со следующими характеристиками:

- CPU - Intel Core i5-8265u (4/8 @ 1.60 GHz / 3.90 GHz)
- RAM-8GB@2400MHz
- OS - Linux Mint 20.3 64-bit

ЗАКЛЮЧЕНИЕ

В ходе дипломной работы был доработан интерпретатор формул *TeX* путем введения статической проверки типов (в том числе типов пользовательских функций) во время стадии семантического анализа.

Программа интерпретатора выполняет чтение входного файла, исполняет стадии лексического, синтаксического и семантического анализа, и в случае корректности содержимого входного файла, производит целевые расчеты и записывает результат в выходной файл.

При внесении изменений не пострадал существующий функционал, что было проверено путем тестирования программы при различных входных данных, использующих разные синтаксические конструкции и типы данных. Замеры производительности показали, что состоятельное ускорение программы может быть выявлено только на стресс-тестах, использующих входные файлы с большим количеством команд во время стадии выполнения, в то время как в случае небольших входных данных производительность практически не отличалась. В таких случаях средний медианный прирост производительности составил 10%.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Курсовые проекты Натальи Стрельниковоǐ и Нурсултана Эсенбаева [Электронный ресурс] URL: https://github.com/bmstu-iu9/tex-preprocessor (дата обращения: 02.04.2022)

2. Барендрегт Х. Лямбда-исчисление. Его синтаксис и семантика. — М.:Мир, 1985 (дата обращения: 17.04.2022)

3. Бенджамин Пирс Типы в языках программирования. — М.:Мир, 2010 (дата обращения: 29.05.2022)

4. A tutorial implementation of a dependently typed lambda calculus [Электронный ресурс] URL: https://www.andres-loeh.de/LambdaPi/LambdaPi.pdf (дата обращения: 04.06.2022)

5. The Hindley-Milner Type System [Электронный ресурс] URL: https://web.mit.edu/6.827/www/old/lectures/L07-Hindley-Milner2Print.pdf (дата обращения: 16.04.2022)

6. Bidirectional Typing Rules: A Tutorial [Электронный ресурс] URL: https://davidchristiansen.dk/tutorials/bidirectional.pdf (дата обращения: 17.05.2022)

7. Bidirectional Typing [Электронный ресурс] URL: https://arxiv.org/pdf/1908.05839.pdf (дата обращения: 10.05.2022)

8. Strict Bidirectional Type Checking [Электронный ресурс] URL: http://adam.chlipala.net/papers/StrictTLDI05/StrictTLDI05.pdf (дата обращения: 11.04.2022)

9. Lecture Notes on Bidirectional Type Checking [Электронный ресурс] URL: https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf (дата обращения: 21.05.2022)

10. Bidirectional Type Inference in Programming Languages [Электронный ресурс] URL: https://homepage.cs.uiowa.edu/~cwjnkins/assets/Jen18_Qualifying-Exam.pdf (дата обращения: 05.06.2022)

11. Hindley-Milner Type Inference implemented in Python [Электронный ресурс] URL: https://github.com/rob-smallshire/hindley-milner-python (дата обращения: 29.04.2022)

12. Dependently Typed Lambda Calculus [Электронный ресурс] URL: https://github.com/ilya-klyuchnikov/lambdapi (дата обращения: 01.06.2022)

ПРИЛОЖЕНИЕ А

Файл *preproc.tex* с определениями для препроцессора TeX

```
\usepackage{ifthen}
\usepackage{amsmath}
\usepackage{color}
\usepackage{mathtools}
\usepackage{tikz}

\newcommand{\placeholder}[2][]{\ifthenelse{ \equal{#1}{} }
                                {\color{blue}#2}
                                {\color{blue}#2 \cdot #1}}
\newcommand{\ifexpr}[1]{{\bf if\enspace}#1\enspace}
\newcommand{\when}{{\enspace\bf when\enspace}}
\newcommand{\otherwise}{{\enspace\bf otherwise}}
\newcommand{\while}[1]{{\bf while\enspace}#1\enspace}
\newcommand{\true}{true}
\newcommand{\false}{false}
\newcommand{\abs}[1]{|#1|}
\DeclarePairedDelimiter{\ceil}{\lceil}{\rceil}
\DeclarePairedDelimiter{\floor}{\lfloor}{\rfloor}

\newenvironment{preproc}
{\color{red}\begin{equation*}\begin{array}{l}}
{\end{array}\end{equation*}}

\newenvironment{block}
{\begin{array}{|l}}
{\end{array}}
```

```
\newenvironment{caseblock}
{\begin{array}{|l}}
{\end{array}}

\newcommand{\range}[3][]{%
	\ifthenelse{\isempty{#1}}{[#2:#3]}{[#2:#3:#1]}%
}

\newcommand{\graphic}[3]{
\color{blue}
\begin{tikzpicture}
\begin{axis}[
	xmajorgrids,
	ymajorgrids,
	axis lines=middle,
	x label style={at={(axis description cs:0.5,-0.1)},anchor=north},
	ylabel style={above left},
	xlabel=\text{#1(#2)},
]
\addplot[thin, mark = none] coordinates{
	#3
};
\end{axis}
\end{tikzpicture}}
```