

Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления
Лаборатория (OS LAB)

Выпускная квалификационная работа магистра

Поддержка union типов в статическом языке программирования

Автор:

Студент М01-206 группы
Акмаев Алексей Михайлович

Научный руководитель:

Добров Андрей Дмитриевич

Научный консультант:

Бронников Георгий Кириллович



Москва 2024

Аннотация

Поддержка union типов в статическом языке программирования

Акмаев Алексей Михайлович

В данной работе исследуются методы поддержки union типов в статически типизированном языке программирования, а также способы генерации и оптимизации байт-кода для этих типов. Union типы или объединения позволяют переменной хранить значение одного из нескольких указанных типов, повышая безопасность, гибкость и выразительность кода при сохранении преимуществ статической типизации. Работа направлена на реализацию union типов в языке, аналогичном TypeScript. В ходе исследования выполнены задачи по реализации базовых union типов с необходимыми отношениями подтипирования, нормализации объединений, поддержке доступа к полям с одинаковым именем и типом, внедрению литералов в union типы, написанию понижающих трансформаций для корректной кодогенерации и оптимизации байт-кода.

Ключевые слова: union типы; объединения; компилятор; байт-код; нормализация; оптимизация.

Содержание

1	Введение	5
1.1	Описание проблемы	5
1.2	Предлагаемое решение	6
1.3	Фундамент для построения и внедрения решения	7
2	Постановка задачи	8
2.1	Проблематика	8
2.2	Цель работы	8
2.3	Задачи	8
2.4	Обработка результатов	8
2.5	Требования	9
3	Обзор существующих решений	10
3.1	Формализация системы типов на основе Featherweight Java	10
3.2	Сочетание статической и динамической типизации на примере StaDyn . .	13
3.3	Теоретико-множественные типы	14
3.4	Анализ существующих подходов	18
4	Исследование и построение решения задачи	19
4.1	Исследование компонент компилятора	19
4.1.1	Лексический анализ	19
4.1.2	Область видимости	19
4.1.3	Объявления	19
4.1.4	Переменные	19
4.1.5	Привязка переменных	20
4.1.6	Синтаксический анализ	20
4.1.7	AST дерево	21
4.1.8	Анализ имен переменных	21
4.1.9	Анализатор типов	21
4.1.10	Типы сущностей в анализаторе	22
4.1.11	Виды отношений между типами	22
4.1.12	Сигнатуры функций	23
4.1.13	Понижающие трансформации AST дерева	23
4.1.14	Кодогенерация из промежуточного представления	24
4.1.15	Аллокация регистров	25
4.1.16	Выделение регистров для локальных переменных при кодогенерации	25
4.1.17	Разрешение имен переменных	25
4.1.18	Узел промежуточного представления	26
4.1.19	Преобразование в следующее промежуточное представление . . .	26
4.2	Схематичное устройство компилятора	27
4.3	Boxing и unboxing операции	27
4.3.1	Определения	27
4.3.2	Преимущества и Недостатки	28
4.3.3	Примеры в разных языках программирования	28
4.4	Семантика и синтаксис объединений	29
4.4.1	Определение объединений	29
4.4.2	Объединения с пользовательскими классами и доступ к полям . .	29
4.4.3	Объединения с литералами	30
4.4.4	Нормализация объединений	31

4.5	Инструкция для доступа к полю объединения	33
5	Описание практической части	34
5.1	Синтаксический анализ	34
5.2	Анализатор типов	35
5.2.1	Наименьшая верхняя граница	36
5.2.2	Алгоритм подсчета низкоуровневого представления объединения .	36
5.2.3	Алгоритм нормализации объединения	38
5.2.4	Алгоритм отношения идентичности объединения другому типу . .	41
5.2.5	Алгоритм отношения подтипирования	42
5.2.6	Алгоритм отношения присвоения	42
5.2.7	Алгоритм отношения приведения	43
5.3	Кодогенерация объединений	45
5.3.1	Алгоритм понижающей трансформации доступа к полю	45
5.3.2	Алгоритм понижения бинарного оператора	47
5.4	Оптимизация объединений	52
6	Заключение	55

1 Введение

1.1 Описание проблемы

Известно, что разработка хороших, многократно используемых библиотек является очень сложной задачей. В популярных статических объектно-ориентированных языках, таких как Java и C++, наследование и подтипизация (а в последнее время и обобщения в том числе) используются в качестве основных механизмов, способствующих многократному использованию кода. В то время как наследование позволяет одному классу повторно использовать реализацию другого класса, например, объявления переменных и сигнатур методов, подтипирование предназначено для взаимозаменяемости. Под взаимозаменяемостью подразумевается такое свойство сущности, что если объект одного типа может быть использован в определенном месте, то и другой объект, являющийся его подтипом, также может быть использован в том же месте. Взаимозаменяемость может быть также перефразирована как возможность повторного использования контекстов в том смысле, что если некоторый контекст применим к объекту одного типа, то тот же контекст также применим к любому объекту его подтипа. Для ясности дадим определения подтипа и супертипа. Подтип - это тип, являющийся производным от другого типа, который называют супертипом. Подтип наследует свойства и поведение своего супертипа, но также может добавлять дополнительные свойства или переопределять существующие. Подтипирование - это способ выразить, что один тип является специализированной версией другого. Супертип предоставляет общее определение, которое может быть расширено или специализировано с помощью его подтипов. Таким образом, проблемы проектирования, связанные с отношениями наследования и подтипирования, несколько различаются: для наследования необходимо учитывать, как новые классы могут повторно использовать существующую реализацию, а для подтипирования - как объекты могут использоваться в клиентском коде.

В популярных языках связь между подтипами по большей части основана на отношениях наследования. Исключением являются только wildcards в Java 5.0. Может случиться так, что два класса, используемые в схожих контекстах, но с довольно разными реализациями, будут разделены в иерархии классов наследования, что приведет к отсутствию полезного супертипа этих классов. Интерфейсы, как программная конструкция, в Java являются решением этой проблемы: можно определить суперинтерфейс классов схожего назначения, независимо от заданной иерархии наследования, и пользоваться преимуществами подтипирования. Однако интерфейсы не могут быть добавлены после определения класса, поэтому разработчикам библиотек по-прежнему приходится много работать над планированием иерархий интерфейсов перед выпуском библиотеки в релиз. Эта проблема считается существенным ограничением систем типов, основанными на взаимозаменяемости, как в Java.

1.2 Предлагаемое решение

В этой работе предлагается решение — объединение или union типы. Union типы или объединения - это тип данных в некоторых языках программирования, позволяющий переменной хранить значение, которое может быть одним из нескольких различных, в том числе несвязанных между собой наследованием, но фиксированных типов. Только один из типов, входящих в объединение, может быть ассоциирован с переменной в среде исполнения в конкретный момент времени. Они позволяют решить проблему невозможности добавления супертипов к существующим типам, таким как классы и интерфейсы.

На практике различают два вида объединения - тегированный и нетегированный union. Нетегированное объединение можно представить как фрагмент памяти, который используется для хранения переменных разных типов данных. Как только ему присваивается новое значение, существующие данные перезаписываются новыми данными. Область памяти, в которой хранится значение, не имеет внутреннего типа (кроме просто байтов или слов памяти). Однако это значение можно рассматривать как один из нескольких абстрактных типов данных, имеющий тип значения, которое было последним записано в область памяти. Нетегированные объединения обычно довольно ограничены в использовании и представлены только в не типобезопасных языках программирования, таких как C. Тегированное объединение можно рассматривать как тип с несколькими компонентами, каждая из которых должна быть корректно обработана при манипулировании этим типом. Говоря об объединении, по умолчанию будет подразумеваться тегированный union.

Поскольку объединения состоят из существующих типов, они дают возможность определять супертип даже после того, как иерархия классов установлена. Как следует из названия, тип объединения обозначает объединение множества заданных типов, рассматриваемых как наборы экземпляров, которые принадлежат к этим типам, и на уровне среды исполнения ведут себя как наименьший супертип. Объединения могут использоваться не только как механизм полиморфизма в сигнатурах функций и при объявлении переменных, но для прямого доступа к полям объектов при соблюдении некоторых условий, как обычные типы. Фактически, для некоторых типов их объединяющий тип можно рассматривать как интерфейс, который "выделяет" их общие черты, то есть поля с одинаковыми именами и методы с похожими сигнатурами. Предполагается, что объединения могут быть полезны для группировки независимо объявленных классов со схожими интерфейсами, но несвязанными друг с другом никакими отношениями, а также для реализации гетерогенных коллекций, таких как списки, в которых строки и целые числа одновременно могут являться их элементами.

1.3 Фундамент для построения и внедрения решения

Фундаментом для исследования послужит проект по разработке некоторого статического объектно-ориентированного языка программирования с TS-подобным синтаксисом. Назовем этот язык MuTS и впоследствии будем использовать для него это название. В этом проекте также уже присутствует спецификация этого языка, которая в основном и будет определять поведение объединений. Кроме того, нам даны виртуальная машина, на которой будет исполняться исходный код, а также набор инструкций байт-кода (ISA). Упрощенно продемонстрируем схему исполнения программы в данном проекте.



Рис. 1: Общая схема исполнения программы

Основная логика будет реализована непосредственно в компиляторе - часть проекта, осуществляющая семантический анализ, проверку типов и кодогенерацию из исходного кода на MuTS в байт-код, который впоследствии отдается среде исполнения.

2 Постановка задачи

2.1 Проблематика

Разработка качественных и многократно используемых библиотек является довольно сложной задачей. Объединения или union типы в статически типизированных языках программирования решают эту и несколько других значимых проблем. Позволяя переменной хранить значение, которое может быть одним из нескольких указанных типов, объединения повышают безопасность типов, гибкость и выразительность кода, сохраняя при этом преимущества статической типизации. Кроме того, перед нашим статическим языком стоит проблема максимальной совместимости с TypeScript. Поскольку union типы часто используются в TypeScript, например в nullish типах или опциональных полях классов и параметрах функций, необходимо реализовать ту же функциональность и в MyTS.

2.2 Цель работы

Поддержать union типы на уровне компилятора, при этом соответствуя синтаксису и семантике TypeScript.

2.3 Задачи

- Спроектировать и реализовать класс для базовых union типов
(let x: A|B = new A())
- Поддержать доступ к общим полям всех составляющих
- Реализовать нормализацию типов, входящих в объединение
- Перенести необходимый функционал в фазу понижающих трансформаций дерева
- Поддержать литералы в качестве составляющих объединения
- Оптимизировать AST дерево с целью уменьшения байт-кода или ускорения времени исполнения

2.4 Обработка результатов

- Обеспечить достаточное покрытие тестами
- Убедиться, что процент пройденных тестов больше 80
- Привести наглядные графики результатов оптимизации

2.5 Требования

- Соответствовать спецификации языка MyTS
- Генерировать валидный байт-код
- Не допускать просадки перформанса во время исполнения

3 Обзор существующих решений

3.1 Формализация системы типов на основе Featherweight Java

В данной статье вводятся типы объединения для объектно-ориентированных языков, основанных на статически типизированных классах [4]. Целью работы является реализация эффективного использования разнородных коллекций и группировка независимо определенных классов с аналогичными интерфейсами, нивелируя сложность подобных возможностей в Java. Реализуются объединения, которые могут быть представлены группой классов путем формирования их супертипа после определения этих классов. Тип объединения позволяет получить доступ к какому-либо полю или методу, играя роль интерфейса, состоящего из общих свойств классов в него входящих. Также в этой статье формализуется ядро системы типов поверх Featherweight Java и доказывается, что система типов надежна. Хотя и ожидается, что она полезна сама по себе, механизм прямого доступа к элементам можно вполне подвергнуть критике за то, что он в основном зависит от равенства имен свойств классов, что может вполне оказаться случайным совпадением. Стоит отметить следующие способы реализации некоторых механизмов:

- Объединение $A|B$ преобразуется в общий супертип A и B или просто в `Object`.
- `Case` и прямой доступ к элементам выражаются в терминах `instanceOf` и `downcasts`.

Рассмотрим подходы к реализации объединений, которые представлены в этой статье. Первое утверждение заключается в том, что $A|B$ является супертипом как A , так и B . Из этого следует, что для классов A и B допускается приведенное ниже присвоение:

```
A|B un = new A();
```

Более того, $A|B$ является наименьшим супертипом среди супертипов A и B в том смысле, что любой общий супертип A и B также является супертипом $A|B$. Таким образом, такое присваивание также разрешено:

```
C x = un;
```

Здесь предполагается, что классы A и B наследуются от класса C . Главным образом объединения можно интерпретировать как тип-множество, но не как супертип всех его составляющих. Это означает, что тип $A|B$ включает в себя только экземпляры A или B , и ничего больше. В то время как другие супертипы могут включать экземпляры, принадлежащие к классам, отличным от A и B . Также обращается внимание на то, что отношение подтипов не является антисимметричным, как в обычных объектно-ориентированных языках. Существуют два синтаксически различных типа, которые являются подтипами друг друга. Например, $A|B$ и $B|A$ синтаксически различны и являются подтипами друг друга. В статье предоставлены

два вида операций с типами объединения: case-анализ и доступ к свойствам классов. Case-анализ - это конструкция условий, которая разветвляется в соответствии с классом значения тестируемого выражения, известного во время выполнения. Например, в данном ветвлении

```
case un of (A x) { x.foo(); }  
         | (B y) { y.bar(42); y.foo(); }
```

вызывается метод `foo()`, если значение `un` является экземпляром `A` (или одного из его подклассов), или методы `bar()` и `foo()`, если фактический класс является экземпляром класса `B` (или одному из его подклассов). Здесь `x` и `y` имеют конкретный тип соответствующего класса, но статически их тип вычисляется как `A|B`. В этом смысле такую конструкцию можно рассматривать как комбинацию динамического тестирования типов во время исполнения (`instanceOf`) и приведения типов. Таким образом, код можно переписать следующим образом:

```
if (un instanceof A) { A x = (A)un; x.foo(); }  
else { B y = (B)un; y.bar(42); y.foo(); }
```

Одним из преимуществ использования этой конструкции является то, что система типов может проверять полноту условий ветвления на соответствие тестируемому выражению. Фактически, требуется, чтобы тип тестируемого выражения был подтипом объединения типов, представленных в ветвях (в приведенном выше примере, `A` и `B`). Это требование гарантирует, что будет исполнена любая ветвь и ее выполнение завершится успешно. С другой стороны, стандартные системы типов не гарантируют успешность приведения типов во второй ветке. Прямой доступ к элементам позволяет напрямую обращаться к полям объединений, если их компоненты содержат поля с одинаковыми именами. В качестве примера рассматриваются следующие определения `A` и `B`:

```
class A extends C {  
    Int fld1;  
    Int fld2;  
    void foo(Int x) { ... }  
}  
class B extends C {  
    Int fld1;  
    Byte fld2;  
    void foo(Int x) { ... }  
}
```

При таком определении классов, возможен прямой доступ к полю `fld1` переменной `un`, имеющей тип `A|B`:

```
Int i = un.fld1;
```

Более того, даже если типы полей различаются, допускается чтение из поля с общим именем:

```
Int|Byte i = un.fld2;
```

В таком случае возвращается объединение типов полей из разных классов. Однако в нашей работе такой доступ к полям не удовлетворяет спецификации и ограничениям языка.

В свою очередь к вызовам методов предъявляются более строгие требования, поскольку методы с одинаковыми именами могут иметь разные сигнатуры. Вызов метода разрешается только в том случае, если имена, количество аргументов и соответствующие типы аргументов полностью совпадают. Следовательно, такой код отработает корректно:

```
un.foo(new Int(42));
```

Возвращаемые типы у методов могут отличаться, если эти типы являются объектными (не void), также, как при доступе к полю. В дополнение, в статье рассматривается ослабление требований к типам в том смысле, что тип аргумент метода является подходящим, если каждый фактический тип аргумента является подтипом обоих соответствующих формальных типов аргументов. Однако такое ослабление доставляет массу проблем при перегрузке методов. Таким образом, прямой доступ к свойствам классов, входящих в объединение, обеспечивает гораздо более лаконичный способ вызова методов или модификации полей, чем использование case-анализа, когда компоненты объединения содержат элементы с общими именами. С помощью этого механизма тип объединения можно рассматривать как своего рода тип интерфейса, который “вычленяет” общие элементы из его компонент. Ожидается, что этот механизм будет полезен когда объединяются независимо определенные классы со схожими функциональными возможностями. Например, A и B могли быть определены отдельно от класса C. В таком случае общий суперкласс A и B мог бы быть разве что только Object. Даже в таком случае экземпляры этих двух классов могут обрабатываться совместно с помощью объединения A|B, и, более того, запрещается смешивать с ними экземпляры других классов (если только они не являются подклассами A или B). Высказывается мнение, что интерфейсы в стиле Java и типы объединений являются скорее дополняющими друг друга механизмами, нежели конфликтующими.

С одной стороны, явно объявленные интерфейсы полезны для абстрагирования от реализаций классов, а также для улучшения документации, поскольку интерфейс предоставляет не только сигнатуры методов, но и более семантические (или поведенческие) концепции реализующих его классов. Например, “метод sort() действительно должен выполнять сортировку” (если такая функция не предусмотрена языком программирования по умолчанию). С другой стороны, объединения более полезны для предоставления апостериорных интерфейсов для legasy или сторонних классов, над которыми программисты не всегда имеют контроль.

Все идеи, упомянутые выше, могут быть отлично использованы и в нашей работе, однако эффективная реализация прямого доступа к элементам без громоздких ветвлений не достаточно хорошо исследована в данной статье.

3.2 Сочетание статической и динамической типизации на примере StaDyn

StaDyn - это объектно-ориентированный язык программирования, основанный на C# 3.0, который поддерживает как динамическую, так и статическую типизацию [1]. Хотя текущая реализация StaDyn поддерживает большую часть функциональности C#, ее минимальное ядро сосредоточено на формализации того, как включить динамическую и статическую типизацию в один и тот же язык программирования. В ядре StaDyn ссылки на переменные могут быть установлены как статически (по умолчанию), так и динамические, что изменяет способ проверки типов.

Ядро StaDyn собирает информацию о типах во время компиляции, чтобы статически осуществить проверку типов по динамическим ссылкам. Одним из способов это сделать в этой работе использовались типы объединения. Тип объединения $A1|A2$ определяет обычное объединение множества значений, принадлежащих $A1$ и набор значений, принадлежащих $A2$, представляющий наименьшую верхнюю границу значений $A1$ и $A2$. Тип объединения содержит все возможные типы, которые может иметь ссылка. Набор операций (например, добавление, доступ к полю, присвоение, вызов или индексация), которые могут быть применены к типу объединения, определяется каждым типом, входящим в объединения. То есть общее подмножество всех операций, которые разрешены для всех типов в объединении, составляют этот набор. Типы объединений уже были включены в объектно-ориентированные языки, в системы типов, где они были явно описаны [2] или выведены из неявно типизированных ссылок [3]. В этой статье взяли другие правила подтипирования, добавив новое правило динамической типизации. Если тип объединения является статическим, то набор операций определяется так, как было сказано ранее. В том случае, если ссылка на объединение динамическая, проверка типов является менее строгой. На практике это означает, что операция становится применима к объединению, если она применима хотя бы к одному из типов, входящих в него. Если операция не может быть применена к какому-либо типу, будет сгенерирована ошибка типа, даже если ссылка является динамической. В этой новой интерпретации тип объединения $A1|A2$ может представлять собой разную сущность в среде выполнения. В случае статической ссылки он является наименьшей верхней границей значений $A1$ и $A2$, а в случае динамической ссылки - каким-то из типов, входящих в объединение.

Проанализировав результаты данной статьи, можно заключить, что подход смешанной типизации позволяет языку программирования StaDyn повышать производительность динамической типизации во время выполнения и гибкость статической типизации. Его система типов выполняет вывод типов как из статических, так и из динамических неявных ссылок, чтобы улучшить производительность во время выполнения и статически проверять динамические типы. В то же время информация о типах, собранная компилятором, позволяет взаимодействовать обоим типам кода, используя одну и ту же систему типов. Оптимизация StaDyn основана на статистическом получении информации о типе динамических ссылок. Наибольшая

выгода достигается при выполнении тестов с динамической типизацией. Однако в этой статье помимо всего прочего необходимо формализовать семантику ядра языка и доказать его типобезопасность при использовании статических ссылок.

3.3 Теоретико-множественные типы

В этой статье описывается использование теоретико-множественных типов в языках программирования и излагается их теория [5]. Теоретико-множественные типы включают типы объединения $T1|T2$, типы пересечения $T1\&T2$ и типы отрицания $!T$. В строгих языках имеет смысл интерпретировать тип как набор значений, которые имеют этот тип (например, `Bool` интерпретируется как набор, содержащий значения `true` и `false`). Таким образом, согласно этому предположению,

- $T1|T2$ - это набор значений, которые относятся либо к типу $T1$, либо к типу $T2$;
- $T1\&T2$ - это набор значений, которые относятся как к типу $T1$, так и к типу $T2$;
- $!T$ - это набор всех значений, которые не относятся к типу T .

Теоретико-множественные типы являются полиморфными, когда они включают типовые переменные. Для того, чтобы дать представление о способах программирования, в котором используются использовать теоретико-множественные типы и который описывается в этой статье, рассматривается классическая рекурсивная функция сглаживания, которая преобразует произвольно вложенные списки в список их элементов. В ML-подобном языке с сопоставлением с образцом это может быть определено так же просто, как

```
let rec flatten = function
  | [] -> []
  | h::t -> (flatten h)@(flatten t)
  | x -> [x]
```

Данный код можно интерпретировать следующим образом:

- Функция `flatten` возвращает пустой список `[]`, когда ее аргумент является пустым списком.
- Если аргумент является непустым списком, то она сглаживает начало `h` и конец `t` аргумента и возвращает объединение результатов (обозначается символом `@`).
- Если аргумент не является списком (т.е., первые два пункта не выполняются), функция `flatten` возвращает список, содержащий только этот аргумент.

Функция `flatten` полностью полиморфна: она может быть применена к любому аргументу и, если списки конечны, всегда завершает работу. Хотя семантику функции легко понять, присвоение ей простого и общего полиморфного типа противоречит всем существующим языкам программирования [7], за единственным исключением: `CDuce` [6]. Это связано с тем, что `CDuce` - это язык, который использует полный набор теоретико-множественных связей типов, и тут необходимо они все (объединение, пересечение и отрицание) для определения `Tree(a)`, типа вложенных списков, элементы которых относятся к типу `a`.

```
type Tree(a) = (a\List(Any)) | List(Tree(a))
```

В этом определении используются следующие обозначения:

- Символ “|” означает объединение.
- Символ “\” обозначает разность, то есть пересечение с отрицанием: `T1\T2 = T1&!T2`.
- `List(T)` - это список элементов типа `T`.
- `Any` - это тип любых значений, так что `List(Any)` - это тип любого списка.

Другими словами, `Tree(a)` - это тип вложенных списков, листья которых, то есть элементы, не являющиеся списками, имеют тип `a`. Таким образом, это либо лист, либо список `Tree(a)`. Тогда достаточно просто указать в аннотации `flatten` правильный тип.

```
let rec flatten: Tree(a)!List(a) = function ...
```

Важным моментом является то, что независимо от типа аргумента `flatten`, выражение всегда хорошо типизировано. Если аргумент не является списком, то создается экземпляр `a` в соответствии с типом аргумента. Если это список, то это также вложенный список, и создается экземпляр `a` с объединением типов элементов, не входящих в этот вложенный список. Другими словами, сглаживание может быть применено к выражениям любого типа, и тип, выводимый для такого применения, - это `List(T)`, где тип `T` является объединением типов всех конечных элементов аргумента, причем аргумент, не относящийся к списку, сам по себе является конечным элементом. Например, статически выведенный тип выражения

```
flatten [3 "r" [4 [true 5]] ["quo" [[false] "stop"]]]
```

будет, соответственно, типом `List(Int|Bool|String)`.

Одной из ключевых особенностей полиморфных теоретико-множественных типов, делающей их универсальными, является то, что они включают в себя все три основные формы полиморфизма:

- Параметрический полиморфизм: описывает код, который может работать с любым типом. По сути это свойство семантики системы типов позволяет обрабатывать значения разных типов одинаковым

образом, то есть выполнять один и тот же код для данных различных типов. В этой статье рассматривается только так называемый полиморфизм второго класса (в смысле [8]), когда количественная оценка переменной не может отображаться ниже конструкторов типов или связей типов;

- Специальный (ad-hoc) полиморфизм: позволяет коду работать с несколькими типами, возможно, с разным поведением в каждом случае, как при перегрузке функций. Он обеспечивает единый интерфейс к разнообразному коду для работы с различными типами, которые могут быть несовместимыми, но допустимыми в данном контексте. Например, он позволяет иметь одну и ту же реализацию для разных типов в случае с оператором `+` (сложение `Int` или сложение `String`);
- Полиморфизм подтипов: создает иерархию более или менее точных типов для одного и того же кода, позволяя использовать его везде, где ожидается любой из этих типов.

Также в частном порядке автором были рассмотрены полиморфные теоретико-множественные типы в более общей постановке, показывая, как эти типы позволяют эффективно вводить некоторые функции и идиомы языков программирования. Это было проиллюстрировано на примере условных ветвлений с применением объединений. В языке с типами объединений мы можем вводить точные условные выражения, которые возвращают результаты разных типов. Например,

```
if e then 3 else true
```

имеет тип `Int|Bool` (при условии, что `e` имеет тип `Bool`). Без типов объединения он мог бы иметь приблизительный тип, например, наименьший супертип тип всех типов исходящих из всех веток, или попросту быть неправильно типизированным. Типы объединений могут также быть использованы для структур, подобных спискам, для смешения разных типов на примере ранее продемонстрированного выражения `flatten`, которое вернуло список типа `List(Int|Bool|String)`.

Это делает объединения незаменимыми для разработки систем типов для существующих нетипизированных языков: примером может служить их включение в `Typed Racket` [9], которое позволяет автоматически добавлять аннотации к статически проверяемым типам на диалекте `Scheme` и в `TypeScript` [10], и `Flow` [11], которые расширяют `JavaScript` за счет статической проверки типов.

В своей статье автор попытался рассмотреть многочисленные преимущества и способы использования теоретико-множественных типов в программировании. Теоретико-множественные типы иногда являются единственным способом ввода некоторых конкретных функций, иногда таким же простым, как функция сглаживания, описанная в начале. Это происходит потому, что теоретико-множественные типы предоставляют подходящий язык для описания многих нетрадиционных, но нередких шаблонов

программирования. Это подтверждается тем фактом, что потребность в теоретико-множественных типах естественным образом возникает при попытке соответствовать системам типов в динамических языках: объединение и отрицание становятся необходимыми для понимания природы ветвления и сопоставления с образцом, пересечения часто являются единственным способом описания полиморфного использования некоторых функций, в определении которых отсутствует единообразие, требуемое параметрическим полиморфизмом. Развитие таких языков, как Flow, TypeScript и Typed Racket, является хорошим доказательством этого современного тренда.

Автор также показал, что даже при использовании теории множеств типы не всегда доступны программисту, они часто присутствуют на метаязичном уровне, поскольку предоставляют базовые инструменты для точного ввода некоторых программных конструкций, таких как варианты типов и сопоставление с образцом. Теоретико-множественные типы предоставляют мощный теоретический инструментарий для изучения, понимания и формализации существующих дисциплин, связанных с типами. Автор продемонстрировал это на примере постепенных типов, которые, благодаря теоретико-множественным типам, могут быть поняты как интервалы статических типов, аналогия, которую можно использовать для переосмысления их теории и их практической реализации. Этот обзор, безусловно, неполный. Например, автор почти не говорил о типах XML и XML-программировании, хотя они послужили первой мотивацией для разработки теории семантического подтипирования, а также для разработки и внедрения таких языков программирования, как XDisce и CDisce. Автор также не упоминал, как обращаться с функциями, которые распространены в современных языках программирования, такими как использование абстрактных типов, интеграция которых со структурными подтипами и полиморфизмом может привести к появлению побочных эффектов.

С формальной точки зрения пока не удалось определить уникальный формализм, который сочетал бы неявно и явно типизированные функции, реконструкцию типов пересечений и расширенное использование типизации вхождений. Но исследователи не так уж далеки от этого. С практической точки зрения требуется еще больше работы. Параметрический полиморфизм с теоретико-множественными типами подразумевает генерацию и разрешение ограничений. У этого есть несколько недостатков. Во-первых, из-за наличия объединений и подтипов, решение задач по ограничению является потенциальным источником вычислительного взрыва, с которым пока еще не очень хорошо справляются. Во-вторых, устранение ограничений затрудняет генерацию информативных сообщений об ошибках в случае нарушения работы программы, а также печать выведенных типов в форме, легко понятной программисту.

3.4 Анализ существующих подходов

В данном разделе были представлены различные подходы к типизации в объектно-ориентированных языках программирования. Описание начинается со статьи о формализации системы типов на основе Featherweight Java, в которой вводятся типы объединения для управления гетерогенными коллекциями и группировки независимо определенных классов с аналогичными интерфейсами. Это позволяет обеспечить эффективное использование гетерогенных коллекций и уменьшить сложность подобных возможностей в языке Java. Далее обзор переходит к рассмотрению языка программирования StaDyn, который поддерживает как динамическую, так и статическую типизацию. Здесь описывается смешанная типизация, где типы объединения играют важную роль. Этот подход позволяет языку StaDyn повысить производительность динамической типизации во время выполнения и гибкость статической типизации. В заключительной части рассматривается использование теоретико-множественных типов в языках программирования. Здесь описывается теория таких типов, как объединения, пересечения и отрицания. Данная статья приводит примеры использования теоретико-множественных типов, включая рекурсивную функцию сглаживания списков. Она также подчеркивает важность использования таких типов для эффективной типизации в существующих и разрабатываемых языках программирования.

В целом, были рассмотрены различные методы и концепции типизации, которые могут быть применены для повышения гибкости, производительности и безопасности в объектно-ориентированных языках программирования. Концептуально, подход к реализации объединений довольно схож в разных исследованиях, особенно это касается подтипирования. С небольшими корректировками данные подходы будут использованы и в нашей работе.

4 Исследование и построение решения задачи

Для того, чтобы внедрить наше решение в существующий проект по разработке компилятора MuTS, сначала необходимо тщательно изучить его внутреннее устройство. Составим обзор компонент компилятора для понимания, как сделать наше решение качественной и логичной частью всего проекта.

4.1 Исследование компонент компилятора

4.1.1 Лексический анализ

Этот компонент преобразует исходный код в последовательность токенов. Входные данные должны быть корректной строкой UTF8. Токены могут быть литералами, знаками препинания или ключевыми словами. Поскольку в JavaScript существуют контекстуальные ключевые слова, например, “static” - это ключевое слово внутри тела класса, но в других местах оно просто идентификатор, токены содержат дополнительное поле `keywordType`, которое всегда соответствует соответствующему ключевому слову, независимо от реального типа токена. Поскольку на этом уровне могут возникать синтаксические ошибки, лексический анализатор может генерировать соответствующие ошибки.

4.1.2 Область видимости

Структуры областей видимости - это сущности, в которых хранятся переменные. Каждая область видимости имеет родителя - область по вложенности выше, все объявления классов, переменных и функций, которые хранятся в специальной таблице переменных. Эта таблица содержит строку в качестве ключа и структуру переменной в качестве значения.

4.1.3 Объявления

Объявления типа `var a` или `let b` во время синтаксического анализа преобразуются в структуры объявлений, которые как раз добавляются в специальную таблицу, описанную выше. Каждое объявление знает имя и узел AST дерева, к которому оно относится.

4.1.4 Переменные

Структура переменной содержит в себе структуру объявления, к которой она привязана, область видимости, в которой она находится, а также фактический статический тип переменной и различные вспомогательные флаги. Переменные не создаются по умолчанию. Объявления преобразуются в переменные, если они проходят проверку в рамках области видимости. Тип переменной неизвестен во время синтаксического анализа и заполняется позже анализатором типов.

4.1.5 Привязка переменных

В компиляторе существует специальный компонент, который создает и проверяет все привязки объявлений к области видимости. Эти привязки, как было сказано ранее, хранятся в таблице переменных. Сам по себе этот этап не является отдельным анализом. Каждая проверка привязки запускается в процессе синтаксического анализа. В настоящее время триггерами могут быть:

- Создание новой области видимости.
- Добавление объявления в текущую область видимости. Если в результате проверки добавить привязку не получается, возникает синтаксическая ошибка.

4.1.6 Синтаксический анализ

Синтаксический анализ осуществляет специальный компонент компилятора, который называется Парсер. Этот анализ логично происходит после лексического разбора исходного кода программы. Парсер является одним из основных компонент и работает параллельно с этапом привязки переменных. Синтаксический анализ является однопоточным, и входные данные обрабатываются только один раз. Для парсинга выбран синтаксический анализатор LR(1), поэтому он видит только следующий токен, но может заглянуть в следующую точку кода. Однако из-за новых возможностей стандарта ES2015 список параметров лямбда функций и шаблоны деструктурирования больше не могут быть корректно прочитаны, заглядывая только на один токен вперед. В этих сценариях синтаксический анализатор работает в отказоустойчивом режиме, что означает, что он следует менее строгой грамматике, чем стандартная. Всякий раз, когда закрывающий токен для этих языковых элементов найден или не найден, выполняется обход построенного AST дерева и проверка его правильности в соответствии с правилами грамматики. По мере того как AST строится во время синтаксического анализа, также создается дерево областей видимости. Как только парсер обрабатывает очередной блок кода или функцию, запускается этап привязки переменных, который создает для них новую область видимости и привязывает их к ней. Поскольку время жизни этих областей совпадает со временем жизни соответствующих узлов AST дерева, структуры представляющие узел также сохраняют у себя эти области видимости. Всякий раз, когда считано объявление переменной, выполняется проверка привязки. Таким образом, в общем случае синтаксический анализатор уведомляет инструмент проверки привязок только о том, что он обнаружил новое начало области видимости или объявление новой переменной.

4.1.7 AST дерево

ASTNode - это базовый класс всех узлов, сгенерированных парсером. Узел AST хранит в себе данные о позиции в исходном коде, родительский узел и другие атрибуты, которые добавляются дочерним классам при наследовании.

4.1.8 Анализ имен переменных

После того, как исходный код обработан парсером, AST проходит анализ имен переменных. После него создается класс программы, содержащий AST дерево, позиции переменных в исходном коде и некоторые метаданные. Главной целью этого анализа является определение типа переменной для обеспечения эффективной многопоточной компиляции без блокировки, не считая планирования потоков. Переменная может быть локальная или лексическая. Во время обхода AST дерева мы ссылаемся на уже сгенерированные области видимости, присвоенные statement узлам, чтобы определить, в какой области мы на самом деле находимся. Всякий раз, когда мы оказываемся в узле, представляющий собой имя (идентификатор) переменной, анализатор пытается разрешить ее тип из текущей области. Если у переменной нет конфликтов с какой-либо другой переменной из области видимости (чаще всего это область видимости функций или цикла), переменная объявляется как локальная. В противном случае она получает лексический индекс из ближайшей области видимости и помечается как лексическая. Каждый раз, когда из области видимости запрашивается лексический слот на переменную, область становится лексической. Это означает, что во время компиляции в начале функции должно быть создано так называемое лексическое окружение. В этом анализе мы также определяем, является ли локальная переменная внутри объявления цикла частью замыкания внутри его тела. Результат этого анализа определяет, должны ли цикл или его объявление быть лексическими или нет. Таким образом, перед переходом непосредственно на стадию кодогенерации AST обрабатывается только один раз. Каждая область видимости содержит в себе информацию о том, нуждается ли она в лексическом окружении или нет, и каждая переменная знает, является ли она лексической или нет.

4.1.9 Анализатор типов

Этот компонент семантически анализирует код, используя AST дерево, области видимости и переменные. Анализатор обходит AST дерево и проверяет каждый узел, используя определенную виртуальную функцию, перегруженную для всех узлов по-своему. Когда анализатор обнаруживает узел с объявлением какой-либо переменной, он выполняет ее поиск во всех областях видимости по степени их вложенности друг в друга. Как только найдено объявление этой переменной в какой-то области видимости, анализатор присваивает тип выражения к узлу дерева, если он задан явно, либо использует для этого вывод типов.

- При незаданной аннотации, тип объявленной переменной выводится с помощью ее определения (выражения-инициализатора).
- При объявлении функции анализатор создает для нее сигнатуру, которая состоит из параметров в заданном порядке и типа возвращаемого значения. Этот тип в свою очередь выводится из явно указанной аннотации или типа выражения, следующего за `return` в теле функции.
- При объявлении интерфейса анализатор создает объектный тип, который хранит в себе все поля и их типы, а также сигнатуры методов и конструкторов интерфейса. С точки зрения анализатора интерфейс и класс практически ничем не различаются (только способом хранения супертипов).
- При объявлении псевдонима типа (`type alias`) анализатор использует тип, который этому псевдониму и присваивается.

4.1.10 Типы сущностей в анализаторе

Как только тип объявления вычислен, ему присваивается значение `ts`-типа объявленной переменной. Используя специальную структуру для хранения этого `ts`-типа, анализатор может проверять на валидность выражения присваивания, бинарные или унарные операции, вызовы функций и конструкторов, доступы к полям класса и наследование. Важно отметить, что `statement` не создает тип, это делают только выражения.

- `Statement` проверяется только семантически. Например, если `statement` проверяется на наличие у него выражения-условия, которое имеет тип `void` вместо `boolean`, анализатор сообщает об ошибке, поскольку выражения типа `void` не могут быть проверены на истинность.
- Выражения, с другой стороны, по своей сути порождают типы. Например, бинарное выражение `5 + 6` порождает числовой тип.

Стоит отметить, что виртуальная функция проверки, переопределенная для всех узлов дерева, всегда возвращает значение `nullptr` для `statement` узлов и тип сущности для обычных выражений.

4.1.11 Виды отношений между типами

В определенный момент во время семантического анализа анализатор типов должен соотнести различные типы друг с другом. Например, если переменной присвоено значение `a = 15`, анализатор сравнивает тип переменной `a` с типом инициализатора, который представляет собой числовое литеральное выражение, приводимое к числовому типу. Если `a` был объявлен как `let a: string`, то это присвоение приведет к ошибке, поскольку тип `string` не может быть присвоен типу `number`. В зависимости от операции, производимых с переменными, существует 4 различных типа отношений:

- **Отношение идентичности:** отношение является истинным, если два сравниваемых типа абсолютно идентичны. Оно используется при повторном объявлении переменной или поля, и это наиболее сильное отношение.
- **Отношение подтипирования:** отношение является истинным, если один тип сущности анализатора является подтипом другого. Оно также истинно и для идентичных типов. Чаще всего (для обычных несложных объектов) подтипирование выражается простым наследованием.
- **Отношение присваивания:** отношение является истинным, если тип с правой стороны может быть присвоен типу с левой стороны. Оно используется при обработке присваиваний, операндов бинарных выражений, наследования (отношение базового и дочернего класса), а также для проверки совместимости типа возвращаемых выражений из `return` с типом возвращаемого значения, объявленного в функциях.
- **Отношение приведения:** отношение является истинным, если тип в левой части может быть приведен к типу в правой части. Оно используется при работе с операторами сравнения и явными приведениями типов. Это самое слабое отношение и оно, вообще говоря, не сильно отличается от отношения присваивания.

4.1.12 Сигнатуры функций

Сигнатуры создаются для функций и методов, и они могут быть явно объявлены в теле интерфейса или класса, например, используя следующий синтаксис: `(a: number, b: string): number`. Существует два типа сигнатур: сигнатуры функций и конструкторов. Сигнатуры функций используются для проверки корректности вызовов функций. Например, если объявление функции с именем `func1` было объявлено с сигнатурой `(a: string, b: string)`, то оно не может быть вызвано с меньшим или большим количеством параметров, чем 2, и аргументы вызова функции должны быть присваиваемы типу параметра сигнатуры в правильном порядке. Таким образом, анализатор выдаст ошибку в любом из этих случаев: `func1(1)`, `func1(1, 2, 3)`, `func1("foo")`, `func1(2, "bar")`. Сигнатуры конструкторов идентичны по своим правилам. Разница лишь в том, что они используются при создании объектов и, соответственно, в выражениях с ключевым словом `new`.

4.1.13 Понижающие трансформации AST дерева

Понижающие трансформации AST дерева - это фаза преобразований, работающая, в основном, после анализатора и перед кодогенерацией, во время которой происходит обход AST дерева и модификация некоторых его

узлов с заменой отфильтрованных выражений на более простые и низкоуровневые конструкции. Стоит уточнить, что не все понижающие проходы выполняются после анализатора. На самом деле, некоторые проходы трансформируют AST дерево и до семантического анализа. У каждого прохода имеются предусловие и постусловие. Как видно из названий, предусловие является триггером для запуска трансформации дерева для конкретного узла, а постусловие проверяет, что преобразование было успешным и не нарушило структуру и инварианты AST дерева. Преимущества внедрения понижающих трансформаций следующие:

- Абстрагирование разнообразной логики – трансформации различных частей дерева могут быть написаны независимо друг от друга
- Упрощение кодогенерации
- Легкость отладки. Дает возможность распечатать и проанализировать состояние AST дерева до и после какой-то определенной трансформации.

Недостатки lowering проходов:

- Увеличение времени компиляции
- Возможная утеря отладочной информации
- Основательный семантический анализ проводится до большинства lowering проходов. После трансформаций не исключено, что состояние AST дерева станет некорректным.
- Подходит только для некоторых задач

Примеры lowering фаз, которые осуществляют трансформацию определенных конструкций языка и AST дерева:

- Обработка лямбда-функций, генерация для них объекта.
- Раскрытие объектных литералов
- Boxing и unboxing переменных
- Замена каких-либо выражений на вызовы специальных функций
- Оптимизации

4.1.14 Кодогенерация из промежуточного представления

Кодогенерация осуществляется параллельно для каждого AST узла. Специальный класс, исполняющий роль этого этапа, контролирует все функции, которые генерируют байт-код или управляют ресурсами.

4.1.15 Аллокация регистров

Формат исполняемого файла требует, чтобы все параметры размещались в конце локальных регистров. Поскольку в данном языке есть инициализация полей и переменных по умолчанию, а также `rest` параметры, от локальных регистров требуется выполнение определенных стандартных действий. Для этого нам нужно загружать соответствующие параметры в регистры. Номер соответствующего регистра зависит от количества используемых локальных регистров, что является циклической зависимостью. Для разрешения этой проблемы, была представлена следующая структура регистра:

локальный n	локальный 1	локальный 0	параметр 0	параметр 1	параметр n
...	65534	65535	65536	65537	...

Таким образом, во время генерации кода выделяемые регистры располагаются в порядке убывания, и при необходимости все `spill-fill` инструкции генерируются сразу в нужном месте. Как только вся кодогенерация завершена, становится известно количество всех выделенных локальных регистров, а аргументы генерируемых инструкций преобразуются следующим образом:

- Локальные регистры отображаются в диапазоне `uint16_t`
- Параметры вычисляются как `UINT16_MAX` - общее количество регистров.

4.1.16 Выделение регистров для локальных переменных при кодогенерации

Каждый раз, когда очередной этап кодогенерации доходит до области видимости блока или функции, соответствующий класс обрабатывает локальные переменные. На этом шаге известно, является ли переменная лексической или локальной. Лексические переменные не затрагиваются, поскольку они уже получили свой лексический индекс во время анализа имен переменных. Поскольку индекс регистра, в котором лежит локальная переменная, считывается или записывается только классом кодогенерации, присваивание регистра на этом этапе безопасно. Всякий раз, когда область видимости заканчивается, эти регистры освобождаются и могут быть повторно использованы позже.

4.1.17 Разрешение имен переменных

При генерации байткода всякий раз, когда требуется разрешить имя переменной (идентификатор), используется тот же метод, что и при анализе имен переменных: начинается поиск имени переменной из текущей области видимости и отслеживается, сколько областей мы проходим. В зависимости от результатов поиска принимается решение, какой байт-код нам нужно сгенерировать для разрешения:

- Местных
- Лексических
- Модульных
- Глобальных

переменных. К этому времени информация о типе переменной, вычисленной анализатором, уже доступна, но в данный момент не используется.

4.1.18 Узел промежуточного представления

Узел промежуточного представления имеет свой определенный класс в структуре компилятора. Каждая инструкция из архитектуры набора команд (ISA), сгенерированная из специального файла, имеет свою структуру, которая содержит только необходимые операнды. Исключением является только самый базовый класс инструкции, который содержит список операндов каждого типа. Этими типами операндов могут быть:

- Виртуальный регистр - VReg
- Иммидиат - Imm
- Строка - StringView
- Метка - Label

Кроме того, каждый узел промежуточного представления содержит в себе узел AST дерева, чтобы получать информацию о позиции какой-либо сущности в исходном коде для генерации отладочной информации. В конце каждой кодогенерации узлы промежуточного представления будут преобразованы в инструкции другого IR. Во время преобразования:

- Сохраняются строковые операнды, которые позже будут добавлены в таблицу строк сгенерированной программы
- Операнды VReg переназначаются на их реальные регистровые индексы.

4.1.19 Преобразование в следующее промежуточное представление

Этот этап преобразует класс кодогенерации в функциональный класс стороннего промежуточного представления, который уже потом используется для оптимизации и записи всех сущностей в исполняемый файл. Список сущностей, которые отдаются в следующее промежуточное представление:

- Метки
- Try-catch блоки

- Инструкции
- Отладочная информация

Также передаются все общеиспользуемые данные. Такими данными могут быть, например, строки и литералы. Всякий раз, когда класс кодогенерации отдает какие-то данные промежуточного представления этому этапу, заполняется таблица функций с сущностями из следующего промежуточного представления.

4.2 Схематичное устройство компилятора

Кратко проанализировав и описав выше основные компоненты компилятора, которые позволяют трансформировать исходный код на языке MuTS в некое промежуточное представление с дальнейшей оптимизацией и генерацией байт-кода, стоит представить обобщенную схему работы данного компилятора:

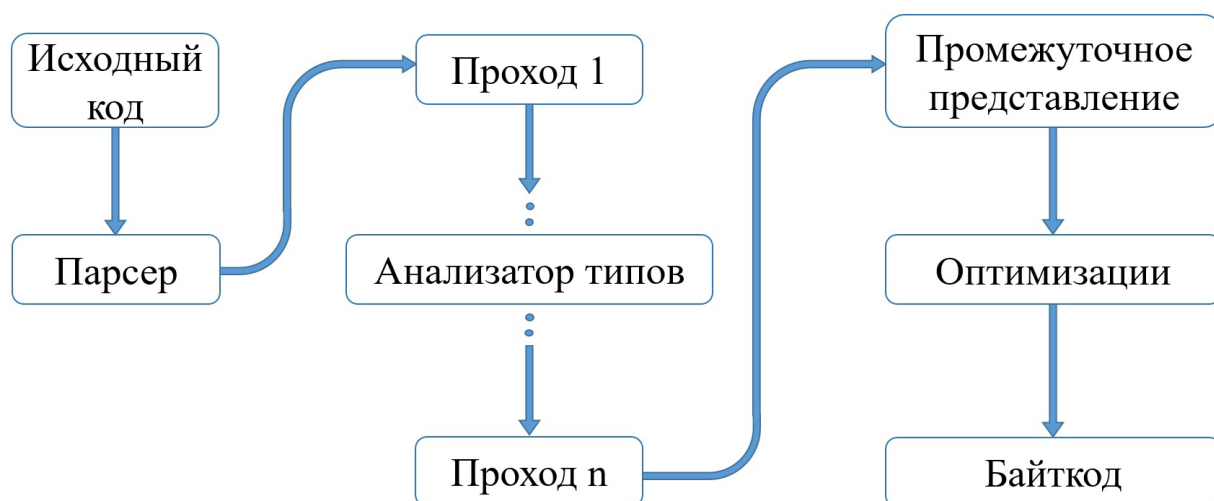


Рис. 2: Схема работы компилятора

4.3 Boxing и unboxing операции

Boxing и unboxing - это важные концепции в языках программирования, особенно в тех, где есть поддержка объектно-ориентированного программирования и системы типов. Эти операции позволяют преобразовывать примитивные типы данных в объекты (boxing) и наоборот (unboxing), что обеспечивает единообразное обращение к данным и их управление.

4.3.1 Определения

Boxing - это процесс преобразования примитивного типа данных в соответствующий ему объект. Например, целочисленное значение типа `int` может быть упаковано в объект `Integer` в языке Java или в объект `System.Int32`

в языке C#. Этот процесс автоматически осуществляется компилятором или средой выполнения. Unboxing, наоборот, представляет собой процесс извлечения значения из упакованного объекта и его преобразования обратно в примитивный тип данных. Например, извлечение целочисленного значения из объекта Integer в Java или из объекта System.Int32 в C#.

4.3.2 Преимущества и Недостатки

Одним из основных преимуществ использования boxing и unboxing является возможность работы с примитивными типами данных в контексте объектно-ориентированного программирования. Это позволяет использовать примитивные типы в качестве аргументов методов, сохранять их в коллекциях и передавать в качестве параметров в методы, требующие объектов. Однако, следует помнить, что процесс boxing и unboxing может быть затратным с точки зрения производительности, особенно при выполнении операций в циклах или при работе с большими объемами данных. Это связано с тем, что при упаковке и распаковке происходит создание дополнительных объектов и процесс их управления.

4.3.3 Примеры в разных языках программирования

Рассмотрим пример использования boxing и unboxing в языке программирования Java:

```
int num = 10;
Integer obj = num;

Integer obj2 = new Integer(20);
int num2 = obj2.intValue();
```

В первом случае на второй строке происходит автоматический boxing, а во втором на четвертой строке - явный unboxing.

В Kotlin также существует поддержка для boxing и unboxing, но благодаря удобным функциям и умному подходу к типизации, эти операции редко требуются. Однако, в некоторых случаях они могут быть полезны. В Kotlin есть возможность использовать inline-функции и inline-классы, которые могут помочь избежать ненужных boxing и unboxing. Компилятор Kotlin может автоматически оптимизировать код, встроив вызовы функций и классов прямо в месте их использования. Хорошая практика в языке способствует минимизации их использования в пользу примитивных типов и эффективных структур данных.

В TypeScript, поскольку это язык сильной типизации, прямого аналога операций boxing и unboxing, как в Java или C#, нет. Однако язык MyTS, используемый в нашей работе, статический и в нем присутствуют такие операции в отличие от TypeScript. Поэтому при присвоении каких-то примитивных значений типу объединения придется использовать лишние вызовы для упаковки или распаковки примитивов. Мы постараемся в данной

работе минимизировать такие дорогостоящие операции и оптимизировать байт-код под среду исполнения.

4.4 Семантика и синтаксис объединений

Для корректной поддержки семантики и синтаксиса TypeScript согласно поставленной задаче, необходимо изучить примеры кода, где используются объединения, и спецификацию языка MyTS. Рассмотрим описание объединений в исследуемом нами статически типизированном языке программирования и сравним их функциональность с объединениями из TypeScript.

4.4.1 Определение объединений

Грамматика union типов задается тем же способом, что и в TypeScript, и может быть записана следующим образом:

```
unionType:
  type|literal ('|' type|literal)*
;
```

Здесь можно заметить, что компонентами объединения могут быть как обычные типы (пользовательские, встроенные, примитивные, и т.д), так и литеральные, представляющие собой конкретные значения встроенных типов. Все составляющие объединения должны быть перечислены через вертикальную черту. Объединение является ссылочным типом, однако в данной работе мы сделаем определенное исключение для оптимизации кода.

Согласно спецификации, ошибка времени компиляции возникает, если тип в правой части объявления union типа приводит к циклической ссылке. Если в объединении используется примитивный тип, то выполняется автоматический boxing, чтобы сохранить ссылочный характер типа. Сокращенная форма объединения типов позволяет определить тип, который имеет только одно значение (литерал), например

```
type UT = 42
```

Рассмотрим различные примеры кода на TypeScript, которые будут валидны и в языке MyTS.

4.4.2 Объединения с пользовательскими классами и доступ к полям

Дадим определения некоторых пользовательских классов A, B и C:

```
class A {
  fieldA: number = 41
  field: number = 5
}
class B {
  fieldB: number = 42
  field: number = 6
}
```

```
class C {
  fieldC: number = 43
  field: number = 7
}
```

То есть в каждом классе есть поле с именем `field` и одинаковым типом `number`, а также другое поле, уникальное для каждого класса. Тогда мы имеем право создать следующие псевдонимы (type alias) на объединения пользовательских классов с примитивным типом `number` и без него:

```
type UT1 = A|B|C|number
type UT2 = A|B|C
```

где `UT1` может быть экземпляром класса `A`, `B`, `C` или числом, а `UT2` может быть экземпляром класса `A`, `B` или `C`. Далее рассмотрим фрагменты кода, которые используют определенные нами типы объединения.

```
let u1: UT1 = new A();
```

Здесь переменной `u1` присваивается тип `UT1`, и ей присваивается экземпляр класса `A`. На этом этапе `u1` конкретно является типом `A`. Рассмотрим доступ к свойству `field`:

```
u1.field
```

TypeScript здесь обязан выдать ошибку, потому что `u1` может потенциально быть числом, а число не имеет свойства `field`. Чтобы безопасно получить доступ к свойству `field`, нужно сузить тип переменной:

```
if (u1 instanceof A) {
  u1.fieldA
  (u1 as A).fieldA
}
u1 = 45
```

Условие `if (u1 instanceof A)` сужает тип `u1` до `A` внутри блока. Внутри этого блока `u1.fieldA` и `(u1 as A).fieldA` оба безопасно получают доступ к `fieldA`. После чего значение `45` типа `number` присваивается переменной `u1`, что допустимо, поскольку `UT1` включает тип `number`. Рассмотрим пример использования `u2`:

```
let u2: UT2 = new C()
u2.field
```

Здесь `u2` имеет тип `UT2`, и ей присваивается экземпляр класса `C`. Поскольку `u2` может быть только `A`, `B` или `C`, доступ к `u2.field` безопасен, потому что все эти классы имеют свойство `field` с одним и тем же типом поля `number`.

Фрагменты кода выше показывают, как объединённые типы TypeScript, сужение типов с помощью `instanceof` и приведения типов (`as`) работают вместе для обеспечения безопасности типов и гибкости кода.

4.4.3 Объединения с литералами

Дадим определения следующих псевдонимов для объединений литеральных типов:

```
type UT3 = 5
type UT4 = 5 | 7 | 9
```

где UT3 — это литеральный тип, который может принимать только значение 5, а UT4 — это объединение, которое может принимать только конкретные значения 5, 7 или 9. Рассмотрим использование типа UT3:

```
let u3: UT3 = 5
u3 = 4
```

Здесь переменной u3, имеющей тип UT3, сначала присваивается значение 5. Это допустимо, поскольку UT3 может быть только литералом 5. Однако на следующей строчке TypeScript выдаёт ошибку, потому что u3 может принимать только значение 5, и попытка присвоить ей 4 недопустима. Далее рассмотрим использование типа UT4:

```
let u4: UT4 = 5
if (u4 == 5) {
  u4 = 7;
} else if (u4 == 7) {
  u4 = 9;
}
```

Здесь переменной u4 присваивается тип UT4, и ей присваивается значение 5. Это допустимо, поскольку UT4 может иметь значения литералов 5, 7 или 9. Далее в условном ветвлении идет проверка значения u4 на равенство какому-либо литералу, после чего оно изменяется в зависимости от результата проверки. Если u4 равно 5, то ему присваивается значение 7. Если u4 равно 7, то ему присваивается значение 9. В любом случае эти присваивания допустимы, так как 7 и 9 входят в допустимый диапазон значений типа UT4. Однако, если мы попытаемся сравнить значение u4 с литералом, которого нет в объединении, то согласно спецификации должна произойти ошибка времени компиляции для такого кода:

```
u4 == 99
```

Эти примеры демонстрируют, как TypeScript использует литеральные и объединённые типы для обеспечения строгой типизации и контроля значений переменных.

4.4.4 Нормализация объединений

Нормализация объединённых типов позволяет минимизировать количество типов и литералов в объединённом типе, сохраняя при этом безопасность типа. Некоторые типы или литералы могут быть также заменены на более общие типы. Формально, объединённый тип $A_1 \mid \dots \mid A_n$, где $N > 1$, может быть сокращён до типа $B_1 \mid \dots \mid B_k$, где $k \leq n$, или даже до не объединённого типа или значения T. В последнем случае T может быть примитивным типом или литералом, изменяющим ссылочную природу типа объединения.

Процесс нормализации предполагает выполнение следующих шагов один за другим:

- **Линеаризация:** Если объединение состоит из двух или более вложенных объединений, они сливаются в одно объединение. Например, `(A1 | A2) | (A3 | A4)` превращается в `A1 | A2 | A3 | A4`.
- **Удаление идентичных типов:** Если в объединении есть несколько одинаковых типов, то они сокращаются до одного. Например, `string | string` превращается в `string`.
То же после boxing: Если один тип в объединении может быть преобразован в другой через boxing, то они заменяются на более общий тип. Например, `number | Number` становится `Number`.
- **Удаление идентичных значений:** Если в объединении есть несколько одинаковых литералов, они сокращаются до одного.
Например, `5 | 5 | 5` превращается в `5`.
- **Доминирование Object:** Если в объединении есть тип `Object`, то все остальные не-nullish типы удаляются.
Например, `777 | Object` становится `Object`.
- Если среди объединённых типов присутствует тип `never`, то он удаляется (`number | never => number`).
- Если в объединении есть непустая группа числовых типов, то в объединении остаётся только самый крупный числовой тип, а остальные удаляются. Любой числовой литерал, который подходит под самый крупный числовой тип в объединении, удаляется. Другими словами, если в объединении есть числовые типы, то остаётся наибольший из них. Например, `int | short | float | 2` становится `float`.
- **Удаление литералов, принадлежащих типам:** Если литерал принадлежит типу, который является частью объединения, то он удаляется. Например, `5 | number` превращается в `number`.
- **Удаление литералов, принадлежащих unboxed типам чисел:** Если числовой литерал принадлежит unboxed типу одного из объединённых числовых типов, он удаляется. Например, `Int | 3.14 | Float` превращается в `Int | Float`.
- **Доминирование базового типа в иерархии классов:** Если в объединении есть класс и его производный класс, то побеждает базовый класс. Например:

```
class Base {}
class Derived1 extends Base {}
class Derived2 extends Base {}
Base | Derived1 => Base
Derived1 | Derived2 => Derived1 | Derived2
```


- Этот шаг выполняется рекурсивно до тех пор, пока не останется взаимно совместимых типов или пока объединённый тип не сократится до одного типа:
 - Если объединённый тип включает два типа A_i и A_j ($i \neq j$), и A_i совместим с A_j , то в объединённом типе остаётся только A_j , а A_i удаляется.
 - Если A_j совместим с A_i , то в объединённом типе остаётся A_i , а A_j удаляется.

Результатом процесса нормализации является либо нормализованный union тип, либо один единственный тип, не являющийся объединением, причем он может быть и не ссылочным типом.

4.5 Инструкция для доступа к полю объединения

Для более эффективной генерации байт-кода, в архитектуре системы команд (ISA), данной нам среде исполнения, предусмотрены специальные инструкции:

- LON_INST (Load Object field by Name Instruction) - загрузка в аккумулятор поля объекта по имени
- SON_INST (Store something into Object field by Name Instruction) - загрузка значения из аккумулятора в поле объекта

Для того, чтобы инструкция в среде исполнения отработала корректно, необходимо сгенерировать специальный класс в байт-коде, который называется `union_member_access`. В этом классе должно быть объявлено поле с тем именем и типом, которое существует для всех компонент объединения. Позже среда исполнения сможет получить доступ к полю объединения на основе этих данных, несмотря на то какой конкретно тип объекта хранится в объединении, причем без приведения к более узкому типу.

5 Описание практической части

В нашей практической части исследования мы столкнулись с необходимостью внедрения объединённых типов в существующий проект по разработке статически типизированного языка программирования MuTS. Развитие статически типизированных языков программирования продолжает оставаться актуальной темой, поскольку такие языки предлагают ряд преимуществ, таких как повышение надёжности и обеспечение более строгой документации кода. При разработке статически типизированных языков программирования одним из ключевых аспектов является поддержка различных типов данных и их комбинаций. Объединённые типы предоставляют механизм для определения переменных, которые могут содержать значения различных типов. Это позволяет программистам более гибко и эффективно работать с данными в коде, обеспечивая при этом строгую типизацию и избегая возможных ошибок времени выполнения.

В данном проекте нам требовалось внедрить поддержку объединённых типов для расширения возможностей системы типов языка программирования. В предыдущей главе мы исследовали различные сценарии использования типов объединения в нашем языке, изучили внутреннее устройство компилятора, а также необходимые сторонние механизмы (такие как boxing/unboxing операции и инструкции загрузки по имени), которые потребуются нам для реализации решения.

На верхнем уровне нам необходимо внедрить наше решение в парсер (синтаксический анализатор), механизм проверки типов и кодогенерации. При необходимости воспользуемся отдельными проходами для модификации AST дерева. Все ключевые алгоритмы ниже будут представлены на псевдокоде в демонстрационных целях.

5.1 Синтаксический анализ

Основная функция, которая парсит пришедший нам union тип, представлена следующим псевдокодом:

```
Function ParseUnionType(FirstType):
    types = []
    types.append(FirstType)
    while NextToken() == "|":
        ReadNextToken()
        nextType = ParseTypeAnnotation()
        types.append(nextType)
    unionType = CreateUnionType(types)
    startPosition = StartPositionOfFirstType(FirstType)
    endPosition = EndPositionOfLastType(types)
    SetRangeOfUnionType(unionType, startPosition, endPosition)
    return unionType
```

Функция начинается с создания вектора типов `types`, который будет содержать все типы, входящие в объединённый тип. Первый тип в объединении добавляется в вектор `types`. Затем функция переходит в цикл

while, который продолжается, пока текущий токен - это побитовый оператор "ИЛИ"(|). Внутри цикла функция считывает следующий токен (переход к следующему токenu). Затем она вызывает функцию ParseTypeAnnotation, которая парсит следующий тип, на который указывает токен после оператора "ИЛИ" и добавляет его в вектор types. Наконец, после выхода из цикла создаётся объект unionType, представляющий объединённый тип, и инициализируется с использованием вектора типов types. Затем устанавливается диапазон объединённого типа от начальной позиции (локации первого типа в исходном коде) до конечной позиции последнего типа в объединении. Наконец, функция возвращает указатель на созданный объект unionType, который является уже частью внутреннего представления программы.

5.2 Анализатор типов

Проверка типов является одним из основных этапов компилятора и происходит после создания синтаксического анализа и построения AST дерева. В анализаторе типов представлены главные структуры промежуточного представления для всех узлов дерева и типов переменных. Нам необходимо написать свой класс для union типа с необходимым функционалом для анализа и дальнейшей кодогенерации. Этот класс должен поддерживать следующую функциональность, как и все остальные типы анализатора:

- Отношение идентичности объединения другому типу
- Отношение подтипирования
- Отношение присвоения
- Отношение приведения

Кроме того, необходимо переопределить много других вспомогательных функций (например строковое представление для вывода ошибок или ассемблерное представление для кодогенерации), а также продумать формат, в котором приходят и обрабатываются наши компоненты объединения.

Было принято решение, что удобнее всего передавать в этот класс уже нормализованные типы, так как логически нормализация должна происходить сразу при создании объединения. Также после нормализации в конструкторе класса будет вычисляться низкоуровневое представление union типа, которое впоследствии будет использоваться в среде исполнения. Для качественной и честной кодогенерации представлять объединение на уровне байт-кода было бы правильным подходом. Однако ввиду срочности производственной задачи было принято решение реализовать более простой вариант низкоуровневого представления, а именно с помощью наименьшей верхней границы. Дадим ее определение.

5.2.1 Наименьшая верхняя граница

Понятие наименьшей верхней границы (LUB - least upper bound) используется там, где необходимо найти единственный тип, который является общим супертипом для набора ссылочных типов. Слово "наименьший" означает, что необходимо найти наиболее специфичный супертип, и не существует другого общего супертипа, который был бы подтипом LUB. Отдельный тип является наименьшей верхней границей сам по себе. В наборе (A_1, \dots, A_n) , который содержит по меньшей мере два типа, LUB определяется следующим образом:

- Для каждого типа в наборе определяется свой набор супертипов $SupA_i$;
- Вычисляется пересечение наборов $SupA_i$. Пересечение всегда содержит объект и, следовательно, не может быть пустым.
- Из пересечения выбирается наиболее специфичный тип.

Ошибка времени компиляции возникает, если какие-либо типы в исходном наборе (A_1, \dots, A_n) не являются ссылочными типами.

5.2.2 Алгоритм подсчета низкоуровневого представления объединения

Низкоуровневое представление объединения необходимо для кодирования union типов в байт-код и вычисляется в основном с использованием наименьшей верхней границы.

```
Function ComputeLowLvlType(unionType):
    preciseType = GetPreciseType(unionType)
    If preciseType is not UnionType:
        return preciseType
    lub = null
    For each type in unionType.ConstituentTypes():
        If type is NullType or lub == type:
            continue
        If lub is null:
            lub = type
            continue
        If type is ObjectType and lub is ObjectType:
            lub = GetLUB(lub.AsObjectType(), type.AsObjectType())
        Else If type is PrimitiveType:
            If lub is ObjectType:
                lub = GetLUB(lub.AsObjectType(), ConvertToObjectType(type))
            Else If lub is PrimitiveType:
                lub = LargestPrimitive(type, lub)
            Else:
                return GetGlobalObjectType()
    Else:
        return GetGlobalObjectType()
return GetNonConstantType(lub)
```

Низкоуровневое представление может являться как ссылочным типом, так и примитивным. Это было сделано для оптимизации объединений, содержащих литеральные типы, и будет описано подробнее в последующих разделах.

Итак, выше на псевдокоде приведена функция вычисления нашего представления объединения и ее можно описать следующим образом:

- Сначала определяется наиболее точный тип для объединения. Наиболее точный тип определяется либо самим типом объединения, либо ограничением на типовой параметр. Если этот тип не является union типом, он возвращается как есть
- Далее инициализируется переменная lub (наименьшая общая верхняя граница) как null
- После чего мы входим в цикл, который перебирает все составляющие типы объединения
- Если текущий тип является NullType или равен lub, то он пропускается
- Если lub еще не установлен, он инициализируется текущим типом и продолжается следующая итерация
- Если оба типа (текущий и lub) являются ObjectType, определяется их наименьшая общая верхняя граница с помощью функции GetLUB.
- Если текущий тип является PrimitiveType, выполняются дальнейшие проверки:
 - Если lub является ObjectType, определяется наименьшая общая верхняя граница между lub (как ObjectType) и текущим типом, преобразованным в ObjectType.
 - Если lub также является PrimitiveType, вызывается функция LargestPrimitive для определения наибольшего примитивного типа между текущим типом и lub.
 - В противном случае, возвращается глобальный объектный тип.
- Если текущий тип не соответствует ни одному из вышеуказанных условий, возвращается глобальный объектный тип
- В конце, возвращается не константный тип, который является наименьшей общей верхней границей для всех составляющих типов union.

Возвращается именно не константное значение, потому что на уровне байт-кода нет представления для литералов. Также низкоуровневое представление будет потом участвовать в кодогенерации и проходах по модификации AST дерева, как примитивный тип.

5.2.3 Алгоритм нормализации объединения

Процесс нормализации объединения происходит в самом начале даже перед подсчетом низкоуровневого представления. Ввиду сложности, рассмотрим алгоритм нормализации в несколько этапов, постепенно погружаясь во все более вложенные функции.

Дадим определение нескольких небольших функций, участвующих в алгоритме:

- Функция `IsNumericUnion`, определяющая, что `union` тип является числовым, возвращает истину в том случае, если все составляющие типы объединения являются маленькими (`unboxed`) числовыми типами или числовыми литералами. В противном случае функция возвращает ложь.
- Функция `BoxTypes`, как видно из названия, преобразует все компоненты объединения в большие (`boxed`) типы.
- Функция `TryAbsorbType` пытается слить воедино два типа, которые в нее пришли. Делает она это путем вызова функции подтипирования для обоих типов, а также довольно непростого анализа удаления литеральных типов, который будет описан чуть позже.

Далее опишем алгоритм линеаризации и удаления идентичных типов на псевдокоде:

```
initialSize = len(types)
for i from 0 to initialSize - 1:
    currentType = types[i]
    if currentType is UnionType:
        otherTypes = currentType.ConstituentTypes()
        add otherTypes to types
        types[i] = null
    else if currentType is NeverType:
        types[i] = null
EliminateAllNullTypes(types)
if not IsNumericUnion(types):
    BoxTypes(types)
for each cmp in types:
    for each it from cmp + 1 to end of types:
        merged = TryAbsorbType(cmp, it)
        if merged is not null:
            if merged == cmp:
                remove it from types
            else:
                remove cmp from types
                cmp = next element after cmp
    else:
        continue to next it
```

В результате работы этого алгоритма получается более оптимизированный и компактный union тип, который может использоваться в дальнейших вычислениях и проверках типов. Опишем подробнее, какие шаги присутствуют в написанном коде:

- **Линеаризация:** Развертывание всех вложенных union типов и удаление типов never. Для каждого элемента в списке types:
 - Если элемент является union типом, его составляющие типы добавляются в конец списка types.
 - Если элемент является типом never, он помечается как null.

Стоит отметить, что рекурсивно раскрываются все вложенные объединения, так как если union тип попал в алгоритм нормализации, значит он был уже кем-то создан. А как известно еще перед созданием объединения все типы, которые в него поступают, проходят такой же процесс линеаризации.

- Далее удаляются все null элементы из списка types, которые были помечены ранее. Также размер списка types уменьшается до количества всех не null элементов.
- **Boxing типов:** Если union не состоит из числовых типов, вызывается функция BoxTypes, описанная ранее.
- **Слияние подтипов:** Итерация по каждому элементу в списке types. Для каждого элемента проверяются все последующие элементы на возможность слияния с помощью функции TryAbsorbType. Если два элемента могут быть объединены, выполняется их слияние:
 - Если слияние возвращает первый элемент, второй элемент удаляется.
 - Если слияние возвращает второй элемент, первый элемент удаляется, и текущая итерация смещается на следующий элемент после текущего.

В более подробном объяснении нуждается процесс слияния литеральных типов, поскольку спецификацией задается требование, что любое значение литерала, попадающее в более расширенный числовой тип, удаляется из объединения. На практике для каждого примитивного типа у нас существует отдельный класс для анализатора типов со своими отношениями подтипирования и идентичности. В этих отношениях в рамках одного примитивного класса невозможно обрабатывать все другие примитивные классы из-за громоздкости и возможных конфликтов с другими компонентами анализатора. Поскольку данные правила подтипирования и удаления литералов задействованы только в объединениях, то логично реализовать необходимый функционал на месте и сделать его частью класса анализатора union.

Следующий псевдокод описывает алгоритм проверки, может ли константный литерал быть присвоен целевому типу в union типе. Алгоритм включает несколько этапов: проверку типов константы и целевого типа, проверку идентичности типов и проверку соответствия константы целевому типу:

```
Function IsLiteralAssignableTo(literal, target):  
    if target == GlobalObjectType:  
        return true  
    if literal is StringType:  
        return target is StringType and  
            (not target is LiteralType or IsIdentical(literal, target))  
    if literal is BigIntType:  
        return target is BigIntType and  
            (not target is LiteralType or IsIdentical(literal, target))  
    if not target is PrimitiveType:  
        if target is UnboxableObject:  
            target = ToPrimitiveType(target)  
        else:  
            return false  
    return IsIdentical(literal, target) or IsLiteralFitsToType(literal, target)
```

Опишем шаги из псевдокода подробнее:

- Если `target` является глобальным объектным типом, возвращается `true`, так `Object` является доминирующим типом над любой компонентой объединения.
- Обработка строковых литералов: Если `literal` имеет тип `StringType`, проверяется, что `target` тоже является строковым типом и либо не является константным, либо идентичен `literal`.
- Обработка типов `BigInt`: Если `literal` имеет тип `BigIntType`, проверяется, что `target` тоже является типом `BigInt` и либо не является константным, либо идентичен `literal`.
- Далее происходит обработка большого (boxed) целевого типа: если `target` не является примитивным типом, значит он является объектом. Для случая объекта проверяется, является ли `target` большим типом. Если да, то `target` заменяется примитивным встроенным типом, полученным с помощью функции `ToPrimitiveType`. Если нет, возвращается `false`.
- В итоге возвращается результат проверки идентичности `literal` и `target` на уровне классов анализатора или результат соответствия литерала целевому типу с помощью функции `IsLiteralFitsToType`

Стоит упомянуть принцип работы функции `IsLiteralFitsToType`. Она берет значение литерала и напрямую сравнивает его с наименьшим и наибольшим значением `target` типа, используя функцию стандартной библиотеки C++ `std::numeric_limits`.

5.2.4 Алгоритм отношения идентичности объединения другому типу

Отношение идентичности определяет с каким другим объектом или типом наше объединение можно считать абсолютно идентичными. Отношение идентичности для ссылочных типов означает, что у этих объектов одни и те же супертипы, один и тот же узел AST дерева, задающий определение типа объекта, а также один и тот же набор типовых параметров и флагов. Отношение идентичности для примитивных типов определяется равенством классов анализаторов (то есть и тот и другой примитивный тип выражаются через `IntType`, например), а также равенством значений, если этот класс представляет собой константу.

Для объединения был реализован свой алгоритм определения отношения идентичности, который можно разбить на две части: идентичность в случае числового объединения и идентичность для объединения ссылочного типа. Рассмотрим алгоритм, написанный на псевдокоде:

```
Function Identical(thisType, otherType):  
    if otherType is UnionType:  
        if EachTypeIdenticalToSomeType(thisType, otherType) and  
           EachTypeIdenticalToSomeType(otherType, thisType):  
            return true  
    if thisType.IsNumericUnion() and otherType is PrimitiveType:  
        return Identical(thisType.lowLvlType, otherType)  
    return false
```

Здесь `thisType` это наш тип объединения, который мы хотим проверить на идентичность другому типу. Рассмотрим реализованные шаги подробнее:

- Если `otherType` также является объединением, проверяется, что каждый тип в текущем union типе связан с каким-то типом в `otherType` и наоборот. Функция `EachTypeIdenticalToSomeType` используется для проверки этой связи. Если обе проверки выполняются, то типы считаются идентичными.
- Если текущий тип (`thisType`) является числовым union типом и `otherType` является примитивным типом (`PrimitiveType`), проверяется их идентичность с помощью низкоуровневого представления `lowLvlType`.
`lowLvlType` подсчитывается при создании объединения, как было описано ранее.
- Если ни одно из условий не выполняется, возвращается результат отношения `false`.

Сравнение с низкоуровневым представлением вместо самого объединения связано с тем, что отношение идентичности для примитивных типов и литералов слишком строгое и нам важно только то, каким типом будет являться наш union по факту при кодировании в байт-код.

5.2.5 Алгоритм отношения подтипирования

Отношение подтипирования, как видно из названия, определяет, является ли один тип супертипом другого. Для ссылочных типов рекурсивно проверяется, выполняется ли отношение идентичности между каким-либо супертипом другого объекта и рассматриваемым ссылочным типом. Подтипирование также выполняется и для идентичных типов. Таким образом, отношение идентичности самое сильное из отношений, которое учитывается во всех остальных алгоритмах отношений. Для объединения был реализован свой метод, определяющий отношение подтипирования и заключается он в том, что все компоненты union типа обходятся по порядку. Для каждой составляющей объединения вызывается метод отношения `IsSuperType(ct, other)`, где `ct` - это компонента union, а `other` - потенциальный подтип. В результате получается худящие правила подтипирования:

- Объединение, состоящее из компонент (A_1, \dots, A_n) является подтипом другого объединения, состоящего из компонент (B_1, \dots, B_k) , если для каждого B_i выполняется отношение подтипирования с A_j , то есть функция `IsSuperType(A_j, B_i)` возвращает истину для любых i и j .
- В частном порядке для объединений, состоящих из литеральных типов, следует, что отношение подтипирования возвращает истину только в том случае, если все литеральные типы одного объединения идентичны без явного приведения типов каким-либо компонентам из другого объединения.

Например, объединение `1|2` является подтипом объединения `1|2|3`, но не является подтипом объединения `1|4`, так как для литералов 2 и 4 не выполняется отношение идентичности.

5.2.6 Алгоритм отношения присвоения

Отношение присвоения определяет, может ли какой-то тип быть присвоен объединению или наоборот. Здесь могут учитываться контекст и неявные преобразования.

Функция `RelationTarget` является основной частью алгоритма и определяет, может ли источник (`source`) быть присвоен целевому типу (`this union type`) в контексте отношений типов. Это выполняется путем проверки, относится ли источник к одному из составляющих типов объединения. Также учитывается необходимость применения `boxing/unboxing` преобразований. Рассмотрим псевдокод этой функции:

```
Function RelationTarget(source):
    boxedSource = BoxType(source)
    if boxedSource != source and not ApplyBoxing():
        return false
    for each ct in ConstituentTypes:
        if IsAssignable(boxedSource, ct):
            if boxedSource != source:
```

```
        node.SetBoxingFlag(boxedSource)
    return true
if boxedSource == source:
    return IsSupertype(thisUnion, boxedSource)
related = false
for each ct in ConstituentTypes:
    if IsAssignable(source, BoxType(ct)):
        if not IsNumericUnion():
            node.SetBoxingFlag(ct)
        related = true
return related
```

Алгоритм может быть разбит на следующие шаги:

- `BoxType(source)`: Упаковываем тип источника, преобразуя его в соответствующий объектный тип (например, `int` к `Integer`).
- Если большой тип отличается от исходного и не требуется применять `boxing`, функция завершает выполнение.
- Далее проходим по всем типам, составляющим объединение и проверяем отношение присвоения `boxed` источника текущему типу из `union`. Если присвоение возможно, устанавливаем флаг `boxing` (если нужно) и возвращаем `true`.
- Если `boxed` источник совпадает с исходным, проверяем, является ли текущий `union` тип супертипом для источника.
- Далее инициализируем переменную `related` как `false` и проходим по всем типам, составляющим `union`.
- Внутри цикла проверяем, можно ли присвоить исходный источник текущему типу из `union` после упаковки. Если присвоение возможно, устанавливаем флаг `boxing` (если `union` тип не числовой) и устанавливаем `related` как `true`.
- В конечном счете возвращаем значение переменной `related`, указывающее на возможность присвоения.

Эта функция будет позже повторно использована для отношения приведения, так как это понятие очень близко к отношению присвоения. В результате она вызывается из другой функции присвоения, которая переопределяется для всех классов анализатора и называется `IsAssignable`.

5.2.7 Алгоритм отношения приведения

Отношение приведения определяет, может ли наше объединение быть преобразовано в какой-либо другой тип с помощью явного приведения. В данной инфраструктуре компилятора, для каждого класса анализатора необходимо переопределить два метода приведения: `Cast` и `CastT`. Функция

Cast определяет, можно ли явно привести рассматриваемый класс анализатора к какому-либо другой типу. В свою очередь CastT действует наоборот и возвращает истину, если какой-либо другой тип может быть явно приведен текущему классу анализатора.

Реализуем данные методы и отобразим их на языке псевдокода:

```
Function Cast(target):
  If target is PrimitiveType:
    If not ApplyUnboxing():
      return false
    node.SetUnboxingFlag(target)
  If InCastingContext():
    For each ct in ConstituentTypes:
      If IsCastable(ct, target):
        return true
    return false
  If IsNumericUnion():
    return IsCastable(thisUnion.lowLvlType, target)
  res = true
  For each ct in ConstituentTypes:
    If not IsCastable(ct, target):
      res = false
      break
  return res
```

Этот метод описывает возможность приведения union типа к целевому типу. Вот что происходит на каждом этапе:

- Если целевой тип (target) является примитивным (PrimitiveType), проверяется возможность применения unboxing. Если unboxing невозможен (not ApplyUnboxing()), функция возвращает false. В противном случае устанавливается соответствующий флаг unboxing для узла AST дерева (node.SetUnboxingFlag(target)).
- Если функция вызвана в контексте приведения (InCastingContext()), проверяется возможность приведения каждой компоненты объединения (ct) к целевому типу (target). Если хотя бы один тип из составляющих типов union (ConstituentTypes) приводим к целевому типу, функция возвращает true. Если ни один из типов не приводим, функция возвращает false.
- Если union тип является числовым, проверяется возможность приведения его низкоуровневого представления к целевому типу.
- Если union тип не является числовым, проверяется возможность приведения каждой компоненты объединения к целевому типу. Если хотя бы одна компонента не приводима к target, то переменная res устанавливается в false и цикл прерывается. В конце функция возвращает значение res.

Пример использования: предположим, что у нас есть union тип $A \mid B \mid C$, где A, B и C — это различные типы. Нам нужно определить, может ли

этот union тип быть приведен к другому типу T. Функция Cast выполняет эту проверку, следуя описанным шагам, и возвращает true, если приведение возможно, и false в противном случае.

Метод CastTarget работает точно таким же образом, что и Assignable, и вызывается внутри себя ту же функцию RelationTarget, которая была описана ранее. Единственная разница заключается в предикате для функции RelationTarget - в случае отношения присвоения этот предикат проверяет Assignable каждой компоненты, а в случае отношения приведения - IsCastable.

5.3 Кодогенерация объединений

В данном разделе будут описаны алгоритмы кодогенерации для объединений, а также различные реализованные проходы компилятора, которые позволяют модифицировать AST дерево и подготовить его к дальнейшим преобразованиям.

5.3.1 Алгоритм понижающей трансформации доступа к полю

В данной секции будет описан алгоритм, позволяющий сгенерировать необходимые сущности в байт-код для доступа к полям объединения. Как было сказано в главе исследования и построения решения задачи, нам необходимо воспользоваться готовыми инструкциями загрузки и выгрузки значения поля любого объекта по имени - LON_INST и SON_INST. Для того, чтобы ими воспользоваться также нужно создать синтетический класс и записать в него информацию об имени поля и его типе.

Однако, прежде чем перейти к описанию понижающей трансформации, стоит упомянуть, что на этапе анализатора типов нам нужно удостовериться, что запрашиваемый доступ к полю валиден. Для этого необходимо:

- Пройтись по всем компонентам объединения
- Для каждой компоненты проверить, существует ли запрашиваемое поле в списке свойств объекта
- Проверить, что тип данного поля во всех компонентах одинаков, иначе выдать ошибку компиляции
- Установить низкоуровневое представление объединения в качестве типа всего выражения доступа к полю (ранее им был непосредственно union тип)

В случае, если проверка прошла успешно, мы доходим до этапа понижающей трансформации.

На следующем этапе реализуется создание и управление синтетическими классами и их свойствами для union типов. Основные действия происходят в трёх функциях. Вот их описание на псевдокоде:

```
Function GetUnionMemberAccessClass():
    foundVar = SearchClassInScope(unionMemberAccessClassName)
    If foundVar != null:
        return foundVar
    ident = CreateIdentNode(unionMemberAccessClassName)
    decl, var = CreateVarDecl(ident)
    classDef = CreateClassDefinition(ident, GLOBAL)
    classDef.SetScope(GetScope())
    classDecl = CreateClassDeclaration(classDef)
    classDef.SetType(GlobalObjectType())
    var.SetScope(classDef.Scope())
    return classDef
```

Данный псевдокод описывает функцию `GetUnionMemberAccessClass`, которая предназначена для поиска или создания специального синтетического класса, используемого для доступа к членам union типов. Вот описание шагов, что выполняет этот псевдокод:

- Поиск существующего класса: в глобальной области видимости (scope) ищется класс с именем `unionMemberAccessClassName`. Если такой класс найден (`foundVar` не null), он возвращается.
- Создание нового класса: если нет, то создаётся новый узел идентификатора (`ident`) с именем `unionMemberAccessClassName`. Далее создаются новое объявление переменной, привязанное к идентификатору, а также новое объявление и определение класса с указанным идентификатором и глобальным модификатором (`GLOBAL`).
- После чего проставляются необходимые области видимости для всех созданных сущностей
- Возвращается определение созданного класса.

Далее рассмотрим функцию `CreateUnionMemberAccessClassProperty`, которая предназначена для создания или поиска свойства в нашем синтетическом классе `unionMemberAccessClass`:

```
Function CreateUnionMemberAccessClassProperty(fieldType, propName):
    unionMemberAccessClass = GetUnionMemberAccessClass()
    classScope = unionMemberAccessClass.GetScope()
    foundVar = classScope.SearchProperty(propName)
    If foundVar != null:
        return foundVar
    fieldIdent = CreateIdentifierForField(propName)
    field = CreateClassPropertyNode(fieldIdent)
    field.SetType(fieldType)
    unionMemberAccessClass.AddProperties(field)
    return var.AsLocalVariable
```

Вот описание шагов, которые выполняет этот псевдокод:

- Получение синтетического класса: вызывается функция `GetUnionMemberAccessClass` описанная ранее.

- Далее получаем область видимости этого класса
- Поиск существующего свойства: в области видимости класса ищется свойство с именем `propName`. Если такое свойство найдено, оно возвращается.
- Создание нового свойства: создаётся новый идентификатор для свойства с именем `propName`.
- Создаёт узел поля класса с созданным идентификатором и устанавливается его тип в соответствии с `fieldType`.
- Созданное поле добавляется в синтетический класс.

Вкратце можно описать так, что если свойство уже существует, оно возвращается. Если свойства нет, оно создаётся и добавляется в класс. Далее из обработчика понижающей трансформации вызывается последняя функция, которая и осуществляет всю работу. Важно отметить, что если получилось так, что у низкоуровневого представления такое свойство уже есть, то данные инструкции для доступа по имени не генерируются и байт-код использует свойства супертипа объединения.

5.3.2 Алгоритм понижения бинарного оператора

На этапе компиляции нам неизвестен фактический тип значения, который может оказаться в объединении во время исполнения программы. Это значение может иметь как тип любой из компонент объединения, так и их подтипы. В результате чего возникает проблема правильного считывания значения при бинарных операциях. Бинарными операторами могут выступать сравнение `==`, сложение, вычитание и прочие функции. Вообще говоря, любые операции, кроме сравнения на равенство, запрещено делать с объединениями. Однако в нашей работе сделано исключение для числовых union типов. Для них остаются легальными такие операции, как сложение, сравнение на больше или меньше и прочие.

Сначала опишем понижающую трансформацию для оператора сравнения на равенство. Забегая вперед, нам необходимо будет сгенерировать несколько `statement`-ов и добавить их в базовый блок, в котором находится искомое нами бинарное выражение, содержащее объединение. Это означает, что нам нужно обрабатывать не какое-то конкретное выражение, а целый блок. Обозначим по какому фильтру мы будем отбирать наше бинарное выражение:

- На этапе анализа типов, обозначаем тип бинарной операции union типом для дальнейшей обработки
- Если левая или правая часть бинарной операции является типом объединения, то возвращается истина. В противном случае возвращается ложь.

Далее для каждого отфильтрованного выражения будет запускать процесс генерации `instanceOf` дерева. Он описывается следующим псевдокодом:

```
Function ReplaceBinaryExprInStmt(block, binaryExpression):
    statement = FindStatement(binaryExpression)
    binaryVarDecl = CreateVariableDeclaration(statement, binaryExpression)
    varDeclId = binaryVarDecl.GetFirstDeclaratorId()
    instanceofTree = null
    binaryExpression.SetLeft(CreateTmpVar(statement, binaryExpression.Left()))
    binaryExpression.SetRight(CreateTmpVar(statement, binaryExpression.Right()))
    isLeftUnion = binaryExpression.Left().IsUnionType()
    unionNode = If isLeftUnion Then binaryExpression.Left() Else binaryExpression.
        Right()
    For each ct in unionNode.ConstituentTypes():
        If not ct.IsUnboxable() AND not ct.IsLiteral():
            continue
        testExpr = CreateInstanceofExpression(unionNode, ct)
        If IsDuplicateBranch(instanceofTree, testExpr):
            continue
        clonedBinaryExpr = binaryExpression.Clone()
        consequentBlock = CreateAssignmentBlock(varDeclId, ProcessBinaryOperands(
            clonedBinaryExpr, ct))
        instanceofTree = CreateIfStatement(testExpr, consequentBlock,
            instanceofTree)
    statement.ReplaceBinaryExpressions(varDeclId)
    InsertAfter(statement, binaryVarDecl, instanceofTree)
    return block
```

Функция `ReplaceBinaryExprInStmt` является одной из двух основных логик понижающей трансформации. Опишем ее по шагам:

- Находим `statement` соответствующий бинарному выражению, чтобы определить, где оно находится в исходном коде.
- Создаем новое объявление переменной (`binaryVarDecl`) для бинарного выражения, чтобы хранить результат вычисления бинарного выражения. Получаем идентификатор первой объявленной переменной (`varDeclId`) для последующего использования.
- Заменяем левый (`Left`) и правый (`Right`) операнды бинарного выражения на временные переменные, которые будут использоваться для хранения промежуточных результатов.
- Определяем, является ли левый операнд `union`-типом (`isLeftUnion`). Устанавливаем `unionNode` как левый или правый операнд в зависимости от того, какой из них является `union`-типом.
- Проходим по всем составным типам (`ConstituentTypes`) объединения, пропуская типы, которые не являются `unboxable` объектами и не являются литералами.
- Для каждого подходящего типа создаем выражение `instanceof`, чтобы проверить, соответствует ли фактическое значение `union` типа какой-

либо его компоненте. Проверяем, не является ли ветка instanceof дублирующейся, и пропускаем её, если это так.

- Клонировем текущее бинарное выражение для последующей обработки.
- Создаем блок присваивания (consequentBlock), который будет выполнен, если instanceof проверка будет успешна. В блоке присваивания обрабатываем операнды бинарного выражения и присваиваем результат идентификатору переменной (varDeclId).
- Добавляем новое if-выражение в дерево instanceofTree, чтобы включить новую проверку instanceof и соответствующий блок присваивания.
- Заменяем все вхождения бинарного выражения в исходном выражении на идентификатор новой переменной (varDeclId).
- Вставляем объявление переменной (binaryVarDecl) и дерево instanceofTree после исходного выражения, чтобы сохранить новые преобразования в коде.
- Возвращаем измененный блок кода (block), который теперь содержит преобразованные выражения и проверки.

Опишем другую важную функцию, участвующую в этом преобразовании - ProcessBinaryOperands.

```
Function ProcessOperandsInBinaryExpr(expr, constituentType):  
    isLhsUnion = expr.Left().IsUnionType()  
    unionNode = If isLhsUnion Then expr.Left() Else expr.Right()  
    otherNode = If isLhsUnion Then expr.Right() Else expr.Left()  
    unionAsExpr = ProcessUnionOperand(constituentType, unionNode, otherNode.GetType()  
    ())  
    otherAsExpr = ProcessOtherOperand(constituentType, otherNode, otherNode.GetType()  
    ())  
    If isLhsUnion Then  
        expr.SetLeft(unionAsExpr)  
        expr.SetRight(otherAsExpr)  
    Else  
        expr.SetRight(unionAsExpr)  
        expr.SetLeft(otherAsExpr)  
    If expr.Left().IsPrimitiveType() and expr.Right().IsPrimitiveType() Then  
        expr.SetOperationType(unionAsExpr.GetType())  
    Else  
        expr.SetOperationType(GlobalObjectType())  
    expr.SetType(GlobalBooleanType())  
    return expr
```

Вот, что она делает по шагам:

- Определение, является ли левый операнд union типом. Это важно для того, чтобы правильно обработать объединение и другой операнд.

- Далее идет обработка `unionNode`, используя другую важную функцию `ProcessUnionOperand`, которая подготавливает тип объединения к сравнению. Также обрабатывается `otherNode`, используя функцию `ProcessOtherOperand`, которая подготавливает другой операнд к сравнению.
- Происходит замена левого и правого операнда в исходном выражении на обработанные выражения в зависимости от того, какой из операндов является `union`-типом.
- Если оба операнда являются примитивными типами, то тип операции тоже становится примитивным. Если хотя бы один операнд не является примитивным типом, тип операции становится `GlobalObjectType`. Это нужно для корректного сравнения `value` и ссылочных типов в среде исполнения.
- Поскольку результат сравнения всегда будет булевым значением, результирующий тип выражения устанавливается как `GlobalBooleanType`.
- В конце возвращается обновленное выражение, которое теперь содержит обработанные операнды и корректные типы.

Для наглядности изобразим обобщенную реализацию алгоритма на схеме:

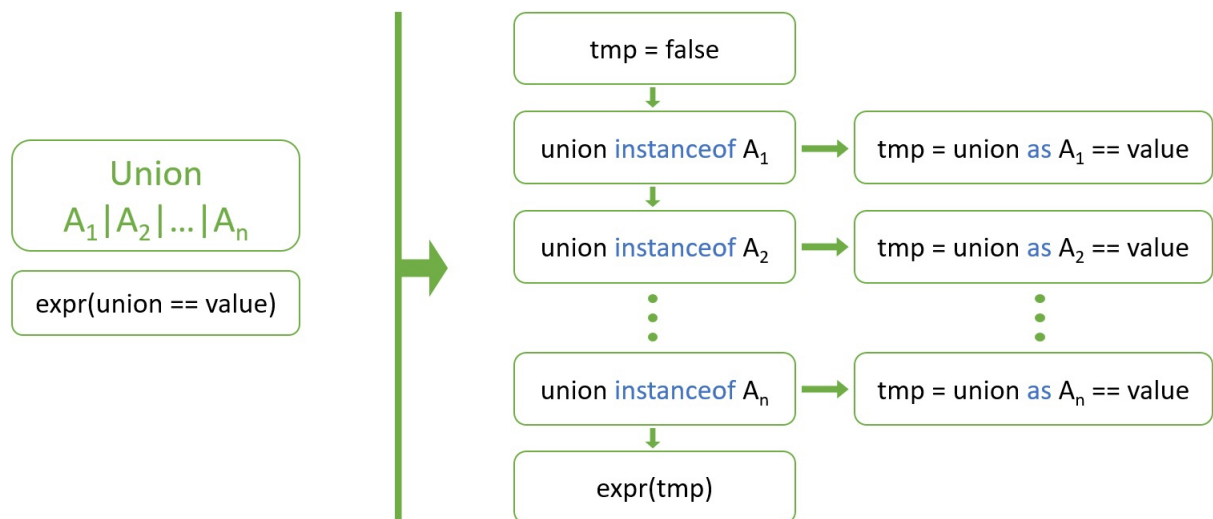


Рис. 3: Алгоритм понижающей трансформации бинарного оператора сравнения

Тут стоит выделить конкретнее, по какой схеме обрабатываются операнды в зависимости от их типа.

В таблице ниже описываются восемь случаев обработки операндов в выражении, где операнды могут иметь разные типы. В таблице указаны типы операндов (`sType` и `other`) и соответствующие модификации для объединения и другого операнда в каждом из случаев:

Обозначения:

cType	other	преобразование
p	p	union as p; None
p	b	union as p; unbox
b	p	union as p; None
b	b	union as p; unbox
p	C	union as box(p); None
b	C	union as b; None
C	p	union as C; box(p)
C	b	union as C; None

- cType: Тип элемента union.
- other: Тип другого операнда.
- p: Примитивный тип.
- b: Упакованный (boxed) тип.
- C: Некоторый неупаковываемый объект.

Вот описание этих случаев:

- cType = примитивный тип (p), other = примитивный тип (p):
union обрабатывается как примитивный тип. Никакие дополнительные преобразования не требуются.
- cType = примитивный тип (p), other = упакованный тип (b):
union обрабатывается как примитивный тип. Другой операнд (other) распаковывается (unbox).
- cType = упакованный тип (b), other = примитивный тип (p):
union обрабатывается как примитивный тип. Никакие дополнительные преобразования не требуются.
- cType = упакованный тип (b), other = упакованный тип (b):
union обрабатывается как примитивный тип. Другой операнд (other) распаковывается (unbox).
- cType = примитивный тип (p), other = некоторый неупаковываемый объект (C):
union обрабатывается как упакованный примитивный тип (box(p)). Никакие дополнительные преобразования не требуются.
- cType = упакованный тип (b), other = некоторый неупаковываемый объект (C):
union обрабатывается как упакованный тип (b). Никакие дополнительные преобразования не требуются.

- `cType` = некоторый неупаковываемый объект (C), `other` = примитивный тип (p):
union обрабатывается как тип (C). Другой операнд (`other`) упаковывается (`box(p)`).
- `cType` = некоторый неупаковываемый объект (C), `other` = упакованный тип (b):
union обрабатывается как тип (C). Никакие дополнительные преобразования не требуются.

5.4 Оптимизация объединений

Использование union типов предполагает возникновение ряда проблем, связанных с производительностью и сложностью анализа типов во время компиляции. Для повышения эффективности и точности анализа union типов применяются различные оптимизации. Эти оптимизации нацелены на улучшение обработки union типов путем уменьшения их сложности и упрощения работы компилятора. Основные задачи оптимизации union типов включают:

- Нормализация: Преобразование сложных union типов в линейную форму для упрощения анализа. Удаление дублирующихся типов внутри union, что снижает количество проверок и операций во время компиляции.
- Упрощение иерархии типов: Выявление и удаление подтипов, которые перекрываются или дублируются. Приведение union типов к наиболее общим формам, что позволяет компилятору выполнять оптимизации на более высоком уровне.
- Оптимизация операций с union типами: Эффективная обработка операций сравнения, присваивания и приведения типов. Специализация и оптимизация часто используемых операций с union типами для улучшения производительности исполняемого кода.
- Преобразование кода для union типов: Преобразование выражений и операций, включающих union типы, в более эффективные формы. Введение вспомогательных структур и механизмов для упрощения работы с union типами в исполняемом коде.

Оптимизации позволяют снизить количество проверок типов и преобразований во время выполнения, что непосредственно влияет на скорость работы программы. Также удаление дублирующих и избыточных типов уменьшает размер исходного кода и конечного бинарного файла. Оптимизированный код с union типами становится более структурированным и понятным, что облегчает его сопровождение и развитие.

Процесс нормализации объединений уже был описан ранее. Основной же оптимизацией данной работы является упрощение кодогенерации для

объединений, содержащих только числовые типы. Поскольку при нормализации в отсутствие нечисловых типов все не литеральные примитивные типы сливаются в один наибольший, единственной возможностью остаться union типом является наличие литералов среди примитивных типов объединения. Соответственно, функция `IsNumericUnion`, реализованная в анализаторе типов, возвращает истину, если наше объединение состоит только из числовых типов, причем среди них есть числовые литералы. И возвращает ложь во всех остальных случаях. Таким образом, в определенных случаях (чаще всего в тернарных выражениях) удастся избежать использования излишних `boxing` и `unboxing` операций, так как они являются очень дорогостоящими вызовами в среде исполнения. Кодирование объединения в примитивный тип дает большие возможности для оптимизаций, а также для JIT и AOT исполнения. Однако увеличивается сложность компиляции и внутренних проверок, а также появляется необходимость в дополнительных понижающих трансформациях, таких как понижение `instanceOf` и `binary` выражений. Однако все эти недостатки нивелируются огромным проростом производительности на отдельных бенчмарках. Итак, прогнав оптимизированную (с `IsNumericUnion`) и не оптимизированную версию (объединение всегда является ссылочным типом) компилятора, получены следующие результаты:

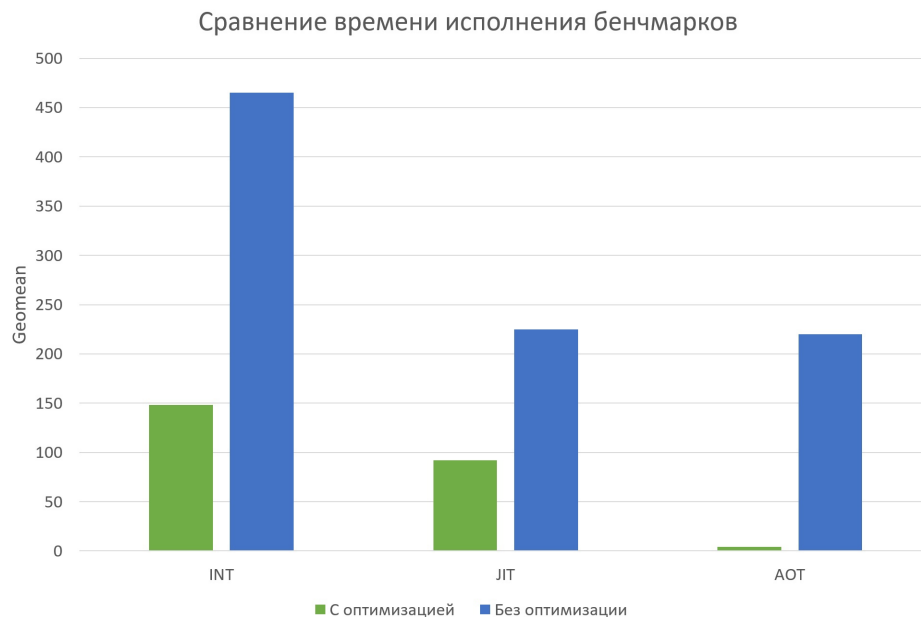


Рис. 4: Сравнение времени исполнения бенчмарков

Как можно видеть из графиков, увеличение производительности составило:

- Для интерпретатора в 3 раза
- Для JIT в 2.4 раза
- Для AOT в 80 раз

Сравнение происходит по геометрическому среднему всех замеров. Стоит отметить, что бенчмарки не случайные и выбирались специальным образом. В них широко используются тернарные выражения, порождающие объединения с литеральными типами, причем в огромных циклах. Также используются функции стандартной библиотеки, в которых также активно задействованы данные операции. Такой выбор бенчмарков связан с тем, что оптимизация сама по себе является довольно узкоспециализированной, и сравнивать тесты без числовых объединений не имеет особого смысла.

6 Заключение

Типы объединений решают несколько ключевых задач в статически типизированных языках, предоставляя гибкость в представлении данных при сохранении безопасности типов. Они улучшают возможность работы с различными формами данных, повышают ясность кода и обеспечивают более надежный и безошибочный код благодаря проверкам на этапе компиляции.

Перечислим основные результаты нашей работы:

- Проектирование и реализация union типов: В анализаторе типов спроектирован и реализован класс для работы с объединениями, обеспечивающий безопасное и удобное использование этих типов в коде. Реализована возможность доступа к общим полям всех составляющих union типов, что позволяет писать более универсальный и многократно используемый код.
- Нормализация типов: Реализованы алгоритмы нормализации типов, входящих в объединение, что позволило устранить дублирующиеся и идентичные типы.
- Поддержка литералов и понижающие трансформации: Реализована поддержка литералов в качестве составляющих объединения, что расширило возможности использования union типов. Перенесены необходимые функциональные элементы в фазу понижающих трансформаций дерева, что, в том числе, улучшило совместимость с TypeScript.
- Покрытие тестами и результаты оптимизации: Процент успешно пройденных тестов составил более 90%, что свидетельствует о высоком качестве реализации. Оставшийся процент обусловлен небольшим количеством недостатков, которые постепенно будут устраняться.
- Графики результатов оптимизации показали значительное уменьшение размера байт-кода и улучшение производительности программ. Улучшение времени исполнения в интерпретаторе и JIT-е составило порядка трех раз.

Реализация поддержки union типов в компиляторе MyTS позволила повысить гибкость и выразительность кода, обеспечив при этом полную совместимость с синтаксисом и семантикой TypeScript. Проведенные оптимизации и тестирование подтвердили эффективность и правильность введенных решений, что делает MyTS более мощным и удобным инструментом для разработки высококачественных библиотек и приложений

Список литературы

- [1] *Francisco Ortin, Miguel García*. Information Processing Letters / Miguel García Francisco Ortin // *Elsevier B.V.* — 2010. — Vol. 111.
- [2] *A. Igarashi, H. Nagira*. Union types for object-oriented programming / H. Nagira A. Igarashi // *Journal of Object Technology*. — 2007. — P. 66.
- [3] *D. Ancona, G. Lagorio*. Coinductive type systems for object-oriented languages / G. Lagorio D. Ancona // *Proceedings of the European Conference on Object-Oriented Programming*. — 2009. — Pp. 2–26.
- [4] *Atsushi Igarashi, Hideshi Nagira*. Union Types for Object-Oriented Programming / Hideshi Nagira Atsushi Igarashi // *Symposium on Applied computing*. — 2006.
- [5] *G., Castagna*. Programming with union, intersection, and negation types / Castagna G. // *The French School of Programming*. — Cham : Springer International Publishing. — 2023. — Pp. 309–378.
- [6] *CDuce*. The CDuce Compiler. <https://www.cduce.org>.
- [7] *Greenberg, Michael*. The Dynamic Practice and Static Theory of Gradual Typing / Michael Greenberg // 3rd Summit on Advances in Programming Languages (SNAPL 2019),. — 2019.
- [8] *Harper, Robert*. Programming Languages: Theory and Practice. — 2006. <http://fpl.cs.depaul.edu/jriely/547/extras/online.pdf>.
- [9] *Sam Tobin-Hochstadt, Matthias Felleisen*. The design and implementation of typed scheme / Matthias Felleisen. Sam Tobin-Hochstadt // *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. — 2008. — Pp. 395–406.
- [10] *Microsoft*. TypeScript. <https://www.typescriptlang.org/>.
- [11] *Facebook*. Flow. <https://flow.org/>.
- [12] *Barbanera F. Dezaniciancaglini M., Deliguoro U.* Intersection and union types: syntax and semantics / Deliguoro U. Barbanera F., Dezaniciancaglini M. // *Information and Computation*. — 1995. — Pp. 202–230.
- [13] *M., Damm F.* Subtyping with union types, intersection types and recursive types / Damm F. M. // *International Symposium on Theoretical Aspects of Computer Software*. — 1994. — Pp. 687–706.
- [14] *Castagna, Giuseppe*. Set-theoretic foundation of parametric polymorphism and subtyping / Giuseppe Castagna, Zhiwu Xu // *6th ACM-SIGPLAN International Conference on Functional Programming*. — 2011. — Pp. 94–106.