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 M01-206 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 способы поддержки union типов в статическом языке программирования, а также варианты генерации байткода для union типов и его оптимизации.

Работа включает: todo

Ключевые слова: union типы; компилятор; MyTS; байткод; нормализация;

Цель работы: реализация union типов в статическом TS-подобном языке программирования с дальнейшей оптимизацией байткода.

Задачи:

- Реализация базовых union типов
- Нормализация union типов
- Поддержка доступа к полям
- Внедрение литералов в union типы
- Написание lowering фаз для корректной кодогенерации
- Оптимизация байткода

Abstract

Support for union types in a static programming language

Содержание

1 Введение

1.1 Описание проблемы

Известно, что разработка хороших, многоразово 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 многоразовому 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 рантайме в конкретный момент времени. Они позволят 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 программировании, 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 рантайма 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 MyTS 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 MyTS в байт-код, который впоследствии отдается среде исполнения.

2 Постановка задачи

2.1 Проблематика

Как уже было сказано ранее, разработка качественных и многократно используемых библиотек это довольно сложная задача. Объединения или union типы в статически типизированных языках 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 проверки типов. Кроме того, перед нашим статическим языком стоит проблема максимальной совместимости с TypeScript. Поскольку union типы часто используются в TS, 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 lowering фазу
- Поддержать литералы в качестве составляющих объединения
- Оптимизировать AST дерево с целью уменьшения байткода или ускорения рантайма

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 байткод
- 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 определеннх 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. Хоть 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 ознаает, 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 соотвествующего 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 legacy 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 проверку 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 примменима к обхединению, 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 Y2;
- `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 способы 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 интерпертировать 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 связок 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 связок 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 разные 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 метауровне, 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 XDuce и CDuce. 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 MyTS, 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, представленными свойством Token::type. Поскольку JS содержит контекстуальные ключевые слова, например, static - это ключевое слово внутри тела класса, но в других местах это простой идентификатор, токены имеют дополнительное поле Token::keywordType, которое всегда соответствует соответствующему ключевому слову независимо от реального Token::type. Поскольку на этом уровне могут возникать синтаксические ошибки, лексер может выдавать соответствующую ошибку.

4.1.2 Область видимости

Структуры binder::Scope - это конструкции, в которых хранятся переменные. Каждая область видимости имеет родителя - область по вложенности выше, все объявления binder::Decl, которые хранятся в таблице переменных Scope::bindings. Эта таблица содержит строку в качестве ключа и переменную binder::Variable в качестве значения.

4.1.3 Объявления

Объявления типа var a или let b во время синтаксического анализа преобразуются в binder::Decl. Каждое объявление знает имя и AST-узел, с которым оно связано.

4.1.4 Переменные

Переменные по умолчанию не создаются. Декларации преобразуются в переменные, если они проходят проверку в рамках области видимости. Структура переменной binder::Variable содержит в себе объявление, из которого она взята, а также имеет checker::Type, который будет представлять фактический статический тип переменной. Этот тип неизвестен во время синтаксического анализа и заполняется позже компонентом Checker.

4.1.5 Binder

Этот компонент создает и проверяет все привязки(bindings) деклараций к области видимости. Сам по себе он 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 она не может добавить привязку, возникает синтаксическая ошибка.

4.1.6 Парсер

Парсер является одним из основных компонент и взаимодействует с binder-ом и лексером одновременно. 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 binder, 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, запускается binder для проверки привязки. Таким образом, в общем случае синтаксический анализатор уведомляет binder 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 идентификатора переменной ir::Identifier, анализатор пытается разрешить ее тип из текущей области. Если у переменной нет конфликтов с какой-либо другой переменной из области видимости binder::VariableScope (чаще всего это binder::FunctionScope, иногда binder::LoopScope), 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 декларация 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 Чекер

Этот компонент семантически анализирует код, используя AST дерево, области видимости и переменные. Чекер обходит AST дерево и проверяет каждый узел, используя виртуальную функцию Checker, перегруженную для всех узлов по-своему. Когда чекер обнаруживает узел с объявлением какой-то 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, чекер 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 инициализатора.
- При объявлении функции чекер создает для нее сигнатуру, которая состоит из параметров в заданном порядке и типа возвращаемого

значения. Оно в свою очередь выводится из явно указанной аннотации типа или выражения, следующего за return в теле функции.

- При объявлении интерфейса чекер создает объектный тип, который хранит в себе все поля и их типы, а также сигнатуры методов и конструкторов интерфейса.
- При объявлении псевдонима типа чекер использует тип, который этому псевдониму и присваивается.

4.1.10 TsType

Как только тип объявления вычислен, ему присваивается значение ts-типа объявленной переменной. Используя структуру checker::Type, чекер 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 вычисляется в тип void, то чекер 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. Вот почему функция ASTNode::Check узла statement всегда возвращает значение nullptr, а функция ASTNode::Check узла выражения всегда возвращает тип, созданный этим выражением.

4.1.11 Отношения

В определенный момент во время семантического анализа чекер должен соотнести различные типы друг с другом. Например, если переменной присвоено значение a = 15, чекер 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, используемой с переменными, существует 3 различных отношения типов:

- 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 присваивания: отношение является истинным, если тип с правой стороны может быть присваивоен 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 приводимости: отношение является истинным, если тип в правой части может быть приведен к типу в левой части. Оно используется при работе с операторами сравнения, приведения типов и в не сильно отличается от отношения присваивания. Это самое слабое отношение.

4.1.12 Сигнатуры функций

Сигнатуры создаются для функций и методов, и они могут быть явно объявлены в теле интерфейса или класса, используя следующий синтаксис: (a: number, b: string): number. Существует два типа сигнатур: сигнатуры функций и конструкторов. Сигнатуры функций используются для проверки правильности 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, чекер 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, соответсвенно, в выражения с ключевым словом new.

4.1.13 Lowering фазы

Lowering преобразование - это фаза трансформаций, работающая после чекера и перед кодогенерацией, во время которой происходит обход AST дерева, преобразуя некоторые его узлы и заменяя отфильтрованные выражения более простыми и низкоуровневыми конструкциями. Стоит уточнить, что не все фазы lowering-а проходят после чекера. На самом деле, некоторые проходы трансформируют AST дерево и до семантического анализа. У каждого lowering 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 lowering проходов перед кодогенерацией следующие:

- Абстрагирует код – различные lowering фазы могут быть написаны независимо друг от друга
- Упрощает кодогенерацию

- Легкость отладки. Дает возможность распечатать и проанализировать состояние AST дерева до и после какой-то определенной трасформации.

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
- Боксинг и анбоксинг переменных
- Преобразование выражений в вызовы рантайм функций
- Оптимизации

4.1.14 Кодогенерация из промежуточного представления

Кодогенерация осуществляется параллельно для каждого AST узла. Класс `Gen` контролирует все функции, которые генерируют байткод 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 шаг кодогенерации попадает в область видимости блока или функции, класс Gen обрабатывает локальные переменные. На этом этапе 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 Gen, 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 ir::Identifier, мы используем тот же метод, что и при анализе имен переменных: начинаем поиск имени переменной из текущей области и отслеживаем, сколько областей видимости мы проходим. В зависимости от результатов поиска принимается решение, какой байткод нам нужно сгенерировать для разрешения:

- Местных
- Лексических
- Модульных
- Глобальных

переменных. На данный момент информация о типе переменной, вычисленной чекером, уже доступна, но в данный момент не используется.

4.1.18 Узел промежуточного представления

Узел промежуточного представления имеет класс IRNode. Каждая инструкция из архитектуры набора команд (ISA), сгенерированная из файла isa.yml, имеет свой класс, который содержит только необходимые операнды. Исключением является ассемблерный класс Ins, который содержит список операндов каждого типа. Этими типами операндов могут быть:

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

Кроме того, их 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 IRNode будут преобразованы в ассемблерные инструкции Ins. 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 Эмиттер

Этот компонент преобразует каждый шаг кодогенерации из класса Gen в ассемблерный класс Function. Список сущностей, которые преобразуются в ассемблер:

- Метки
- Try-catch блоки
- Инструкции
- Отладочная информация

Также сохраняются все общеиспользуемые данные в класс ProgramElement, находящийся в компиляторе. Такими данными могут быть, например, строки и литералы, которые позже записываются в ассемблерный класс Program. Всякий раз, когда генерируется класс Gen, эмиттер объединяет все элементы программы промежуточного представления ProgramElement и заполняет таблицу функций уже созданными ассемблерными классами Function.

4.2 Схематичное устройство компилятора

Кратко проанализировав и описав выше основные компоненты компилятора, которые позволяют трансформировать исходный код на языке MyTS в ассемблерное представление `Program` с дальнейшей оптимизацией и генерацией байткода, стоит представить обобщенную схему работы данного компилятора:

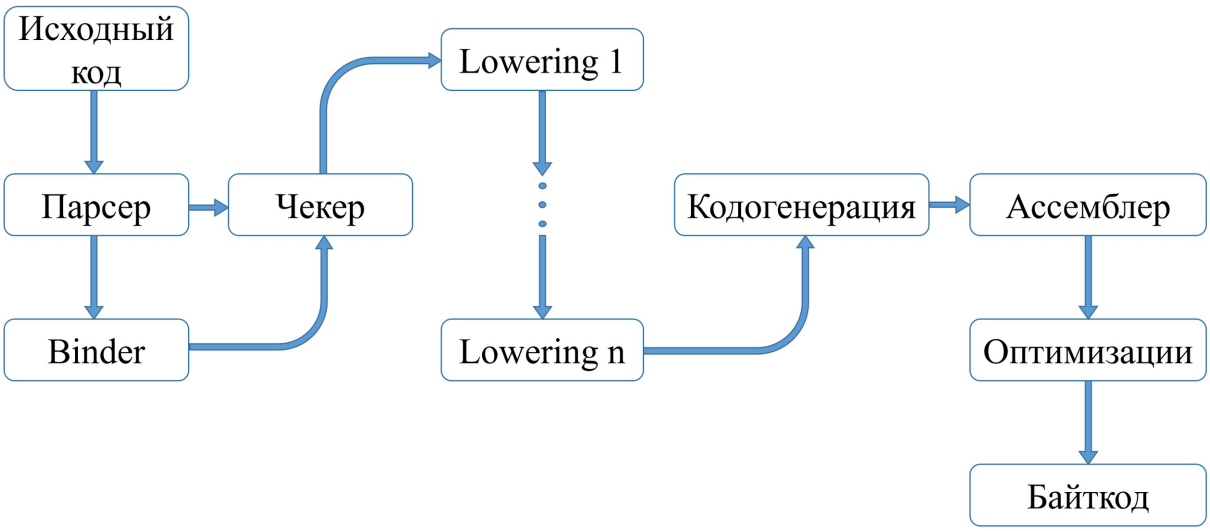

Рис. 2: Схема работы компилятора

todo: описать боксинг анбоксинг

5 Описание практической части

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/память).

6 Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

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. Эти преимущества делают типы объединений мощной функцией для разработчиков, работающих в статически типизированных языках.

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/`.

Приложение

Здесь необходимо написать приложение, которое вы должны придумать самостоятельно.