Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Работа допущена к защите
зав. кафедрой

_____ В.М. Ицыксон

«____» _____ 2019 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Разработка параметризированных типов с переменным набором параметров для языка Kotlin

по направлению 09.04.01 «Информатика и вычислительная техника»
по образовательной программе
09.04.01_15 «Технологии проектирования системного и прикладного программного обеспечения»

Выполнил студент гр. 23541/3 _____ П.С. Кирпиченков

Научный руководитель,
к. т. н., доц. _____ В.М. Ицыксон

Научный консультант,
ассистент _____ М.А. Беляев

Консультант по нормоконтролю,
к. т. н., доц. _____ А.Г. Новопашенный

Санкт-Петербург
2019

РЕФЕРАТ

На 65 с., 35 рисунков, 1 таблица, 1 приложение

ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ, KOTLIN,
ПЕРЕМЕННАЯ АРНОСТЬ

Параметрический полиморфизм широко распространен в современных языках программирования, но лишь некоторые из них позволяют работать с переменным набором типовых параметров. В работе дается классификация видов полиморфизма и рассматриваются существующие языки программирования с поддержкой типовых параметров переменной размерности. Предлагается расширение синтаксиса для языка Kotlin, реализованное в прототипной версии компилятора. По результатам тестирования прототипа дается оценка возможностей и ограничений предложенного подхода.

THE ABSTRACT

65 pages, 35 pictures, 1 tables, 1 appendicies

PARAMETRIC POLYMORPHISM, KOTLIN, VARIABLE ARITY

Parametric polymorphism is widely present in modern programming languages, but only some of them support type parameters of variable arity. In this work we give a classification of polymorphism and make a review of how existing languages deal with variably-sized type parameters. We then suggest a syntax extension for Kotlin, that is implemented in a compiler prototype. We also give our evaluation of the new approach based on the testing results.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Полиморфизм характерен для большинства современных языков программирования высокого уровня. Полиморфные объявления дают возможность писать обобщенный код и избегать дублирования, что в свою очередь упрощает его поддержку: внесение изменений и исправление ошибок. Тем не менее, разные языки программирования по-разному подходят к реализации полиморфизма, а общее множество поддерживаемых видов полиморфизма может отличаться от языка к языку.

Решение о добавлении новых возможностей в язык программирования, как правило, принимается на основе практической потребности, следованию идеям языка и технической возможности реализации. Это справедливо как при создании принципиально новой функциональности, так и при модификации существующих конструкций языка. В настоящее время существуют языки программирования (например, C++, D), позволяющие обобщать полиморфные конструкции на произвольное, не известное в момент объявления количество типов. Подобные объявления имеют ряд важных практических применений, однако их поддержка в других языках программирования не сильно распространена. В языках C++ и D использование полиморфизма с переменным набором типовых параметров возможно благодаря генерации кода для полиморфных объектов в момент компиляции. Цель исследования в данной работе состоит в оценке возможностей для реализации аналогичной функциональности в языке Kotlin, не использующем кодогенерацию для поддержки параметрического полиморфизма.

Дальнейшее содержимое работы организовано следующим образом. В разделе 1 дается классификация видов полиморфизма и рассматриваются существующие подходы к реализации параметрического полиморфизма с переменным набором типовых параметров, а так-

же приводятся практические примеры его использования. В разделе 2 ставятся цели и задачи работы. Раздел 3 содержит описание особенностей языка Kotlin, которые повлияли на дизайн синтаксиса, а также получившийся в результате набор правил по работе с переменным набором типовых параметров. Раздел 4 посвящен описанию изменений в компиляторе, которые потребовались для поддержки дополненного синтаксиса. В разделе 5 рассказывается о процедуре тестирования прототипа, а также дается оценка результатов.

1. ПОЛИМОРФИЗМ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

1.1. Классификация видов полиморфизма

Под полиморфизмом понимается возможность выполнения одного и того же кода с разным смыслом в зависимости от контекста использования. Согласно определению [9], полиморфные сущности предоставляют один интерфейс для объектов разных типов. В качестве примера можно рассмотреть оператор «+». Предположим, что вызов оператора над аргументами числовых типов выдает в качестве результата их сумму в математическом смысле, а вызов над строковыми аргументами производит конкатенацию этих строк. При подобном поведении оператор является полиморфным, поскольку одной и той же записи соответствует операция над разными типами.

В соответствии с классификацией, введенной Карделли [1], существует два подкласса полиморфизма: универсальный (universal) и ограниченный (ad-hoc). Универсальный полиморфизм назван так из-за возможности использовать полиморфный код для потенциально неограниченного множества типов. В свою очередь ограниченный полиморфизм обобщает логику только для фиксированного набора типов. Универсальный полиморфизм включает в себя два подхода: полиморфизм наследования и параметрический полиморфизм. К методам ограниченного полиморфизма относятся перегрузки и приведения типов.

Перегрузкой функции или оператора называется его многократная реализация для разных типов аргументов. При этом код, обрабатывающий один определенный набор типов аргументов, потенциально никак не связан с реализациями для других типов. При использовании такой полиморфной функции по месту ее вызова определяется тип аргументов, после чего происходит поиск подходящей реализа-

ции.

Приведения типов позволяют на основе фиксированного набора правил, определяемых языком программирования, выполнять неявные преобразования типов аргументов в момент использовании функции. Это дает возможность использовать существующую реализацию функции, даже если ожидаемые типы аргументов не совпадают с фактическими, при условии, что последние могут быть преобразованы к нужному типу. Например, в языке JavaScript оператор «+» может вызвать приведение типов: если один из операндов имеет строковый тип, то второй операнд также будет преобразован к строке, после чего будет выполнена конкатенация.

Полиморфизм наследования является одной из ключевых концепций в объектно-ориентированных языках программирования. Его суть заключается в возможности использования имен, определенных для базового типа, во всех типах-наследниках внутри иерархии (если имя не скрыто из соображений инкапсуляции). Однократная реализация функции для базового класса дает возможность вызвать ее на потенциально бесконечном множестве наследников данного класса.

В свою очередь параметрический полиморфизм позволяет определять полиморфные типы в общем виде с абстрагированием от конкретных типов параметров. В императивных языках програмирования он обычно поддерживается в виде обобщенных типов (generics) или шаблонных типов (templates). Обращение к параметрам происходит через типовые переменные, что позволяет использовать один и тот же код при подстановке разных значений типовых аргументов в момент использования полиморфного объекта. Например, полиморфный список объектов может предоставить реализацию множества операций над элементами списка, абстрагируясь от типа этих элементов. В качестве примера таких операций можно привести взятие элемента по индексу, очистку списка. При задании дополнительных требований к типовой переменной набор реализуемых функций может быть расширен.

Так, если задать ограничение, что тип элементов списка поддерживает сравнение, становится возможным реализовать сортировку списка для любых типов, удовлетворяющих ограничению. В данной работе анализируется возможность расширения синтаксиса языка Kotlin для поддержки нового вида параметрических полиморфных объявлений.

Появившееся в языке Haskell понятие классов типов (type classes) [13] расширило множество методов ограниченного полиморфизма. Ряд языков, таких как Rust и Scala, поддерживают во многом схожий механизхм типажей (traits). Классы типов создавались как замена перегрузкам с большим контролем над типами переопределямых операций. В классе типов определяется набор операций над абстрактным типом данных без предоставления реализации этих операций. Реализации создаются независимо в виде экземпляра класса для конкретного типа данных. Так, для арифметических операций может быть создан класс типов `Num` с операциями сложения, умножения и отрицания. Экземпляры класса затем могут быть определены для вещественных и целых чисел, и предоставленные в них реализации операций будут учитывать особенности типа данных, как и в случае с обычной перегрузкой операторов. Подобный подход, помимо контроля над типами, позволяет определять полиморфные функции без экпоненциального роста числа реализаций относительно количества аргументов, допускающих перегрузку. В качестве примера приведем рассмотренную в статье [13] функцию, принимающую три аргумента и возвращающую значения их квадратов. Даже если оператор умножения перегружен только для целых чисел и чисел с плавающей точкой, требуется восемь экземпляров функции для полиморфной реализации с помощью перегрузок. Использование классов типов позволяет обойтись единственной реализацией функции возведения трех чисел в квадрат, определенной относительно класса типов `Num`.

Альтернативным критерием для классификации методов полиморфизма может служить время выбора используемой реализации

функции или оператора. С данной точки зрения все методы делятся на статические, для которых выбор реализации возможен на этапе компиляции, и динамические, при использовании которых подходящая реализация определяется на этапе выполнения. К статическим методам относятся методы параметрического полиморфизма, перегрузки и классы типов. К динамическим относится полиморфизм наследования и приведения типов. Итоговая классификация по двум рассмотренным критериям приведена в Таблице 1.1.

Таблица 1.1

Классификация видов полиморфизма

	Универсальные	**Ограниченные**
Статические	Параметрический полиморфизм	Перегрузки Классы типов
Динамические	Полиморфизм наследования	Приведения типов

1.2. Предпосылки для использования параметрического полиморфизма с переменным набором параметров

В параметрически полиморфных объявлениях для обозначения типов, являющихся параметрами, применяются типовые переменные. Обычная типовая переменная заменяет один тип, который подставляется вместо этой переменной при использованиии. Если обобщение возможно относительно нескольких типов, для каждого из них требуется отдельная типовая переменная. Но иногда код может быть обобщен относительно заранее неизвестного количества типов. Такие случаи невозможно в полной мере выразить с помощью обычных типовых переменных, поскольку их количество жестко задается в момент объявления.

Рассмотрим сценарии, в которых возможность использовать переменный набор типовых параметров может оказаться полезной. Одним

из примеров класса, оперирующего переменным набором типовых параметров, является кортеж. Кортеж представляет собой коллекцию упорядоченных элементов произвольных типов. Типы элементов в кортеже в общем случае различны, но конкретный тип отдельного элемента не важен: важна позиция элемента с таким типом внутри кортежа. Операции над кортежем обобщены как относительно типов элементов (зная позицию, можно достать элемент любого типа), так и относительно их количества (например, итерирование по элементам имеет одинаковый смысл для кортежей разной длины). Использование обычных типовых переменных позволяет обобщить тип отдельных элементов, но не их количество. Для каждой новой длины кортежа нужен новых набор типовых переменных, а значит, и новое объявление.

В языке C++ до появления стандарта C++11 для реализации кортежа разработчики были вынуждены имитировать переменный набор параметров и обходиться существующими конструкциями языка: значениями шаблонных параметров по умолчанию, перегрузками функций, директивами препроцессора. Такие реализации имеют недостатки в производительности, масштабируемости, отладке и поддержке [2]. Рассмотрим пример, в котором определены перегрузки функции `make_tuple`, создающей кортеж из переданных элементов:

```
1  tuple<> make_tuple() {
2      return tuple<>();
3  };
4  template <typename T1>
5  tuple<T1> make_tuple(const T1& t) {
6      return tuple<T1>(t1);
7  };
8  template<T1, T2>
9  tuple<T1, T2> make_tuple(const T1& t1, const T2& t2) {
10     return tuple<T1, T2>(t1, t2);
11 };
```

Рис.1.1. Объявление функции создания кортежа

В примере заданы объявления функции `make_tuple` для кортежа длины 0, 1 и 2 соответственно. Для каждого нового количества типо-

вых аргументов потребуется дополнительное объявление, отличающееся от всех остальных только количеством параметров и аргументов. При такой реализации максимальная длина кортежа имеет фиксированную верхнюю границу, определяемую количеством реализованных перегрузок. Количество перегрузок обычно определяется соображениями производительности и практической потребности.

Альтернативный подход состоит в использовании средств предобработки и кодогенерации для имитации переменного набора параметров. Однородные конструкции определяются относительно некоторого счетчика и повторяются по шаблону. Такой подход позволяет уменьшить объем кода, который необходимо создать вручную, но ценой этого является увеличение времени обработки перед компиляцией, меньшая поддержка средствами анализа кода и, как правило, плохая читаемость исходного кода, используемого для кодогенерации. Кроме того, сохраняется ограничение на максимальное количество параметров, поскольку результат генерации будет содержать ограниченное количество вариантов.

Класс для хранения значения, тип которого допускает один из ограниченного множества вариантов – еще один пример полиморфного объявления, способного получить выгоду от переменного набора типовых параметров. Такой класс может использоваться для выражения типов-сумм в языках без их встроенной поддержки. Например, при создании древовидных структур данных с помощью класса-варианта может быть описан единый тип для всех элементов дерева. Хранимое внутри варианта значение всегда имеет один из двух типов: либо лист, либо узел. Соответственно, тип элемента дерева – вариант с двумя типовыми аргументами: типом узлового элемента и типом листа. Тип варианта полиморфен относительно переменного набора типовых параметров, поскольку их количество фиксируется в момент создания, а не объявления. Вариант – важный пример типа, в котором часть типовых аргументов не представлена в типах хранимых

значений. В примере с кортежем можно было допустить, что тип типовых аргументов вычисляем на основе хранимых значений. Для варианта такой способ получения информации о типовых аргументах невозможен, поскольку значение всегда одно, а допустимых типов – несколько. В языке C++, начиная со стандарта C++17, появилась встроенная поддержка вариантов [7]. В классе std::variant набор допустимых типов реализован с помощью переменного набора шаблонных параметров.

Другой пример, в котором переменный набор параметров был бы полезен, взят из библиотеки Reactive Extensions на языке TypeScript [11].

```
export interface ObservableStatic {
    fromCallback<TResult>(
        func: Function, context: any, selector: Function
    ): (...args: any[]) => Observable<TResult>;

    fromCallback<TResult, T1>(
        func: (arg1: T1, callback: (result: TResult) => any) => any,
        context?: any, selector?: Function
    ): (arg1: T1) => Observable<TResult>;

    fromCallback<TResult, T1, T2>(
        func: (
            arg1: T1, arg2: T2,
            callback: (result: TResult) => any
        ) => any,
        context?: any, selector?: Function
    ): (arg1: T1, arg2: T2) => Observable<TResult>;
// ...
    fromCallback<TResult, T1, T2, T3, T4, T5, T6, T7, T8, T9>(
        func: (
            arg1: T1, arg2: T2, arg3: T3,
            arg4: T4, arg5: T5, arg6: T6,
            arg7: T7, arg8: T8, arg9: T9,
            callback: (result: TResult) => any
        ) => any,
        context?: any, selector?: Function
    ): (
        arg1: T1, arg2: T2, arg3: T3, arg4: T4, arg5: T5,
        arg6: T6, arg7: T7, arg8: T8, arg9: T9
    ) => Observable<TResult>;
}
```

Рис.1.2. Объявление интерфейса ObservableStatic

Интерфейс описывает по сути одну и ту же функцию высшего

порядка для преобразования исходной функции в новую, возвращающую объект `Observable` от результата первоначальной функции. Такое преобразование допустимо для функций от любого количества аргументов с любым типом. Но из-за невозможности выразить это свойство в используемой библиотекой версии языка TypeScript, разработчику приходится определять 10 версий функции и поддерживать только обработку функций с арностью от 1 до 9, а обработку остальных вариантов оставлять без поддержки средствами типизации. Возможность определять типовые параметры произвольной размерности позволила бы одним объявлением функции `fromCallback` от переменного количества аргументов поддержать обработку функций любой арности.

Задача выражения типа произвольной функции актуальна также для языков Java и Kotlin. Стандартные типы для функций в Java 8 ограничены арностью 2, т.к. пакет `java.util.function` содержит только интерфейсы `Function` и `BiFunction` для функций от одного и двух аргументов соответственно. Типы для функций от большего числа аргументов необходимо создавать вручную, при этом в языке нет средств, позволяющих выразить тип функции в виде одного универсального интерфейса для любой арности.

В Kotlin для задания типов функций используется синтаксис функциональных типов. Для примера приведем функцию `filter`, принимающую предикат в качестве аргумента. Ее тип показан на рис. 1.3.

```
1 inline fun <T> Iterable<T>.filter(
2     predicate: (T) -> Boolean
3 ): List<T>
```

Рис.1.3. Функция filter

В процессе компиляции функциональные типы представляются в виде функциональных интерфейсов. Благодаря ним функции высших порядков, объявленные в Kotlin, могут быть вызваны в коде на

Java. Для этого аргумент-функция должен быть экземпляром функционального интерфейса. В компиляторе даны отдельные объявления функциональных интерфейсов для арностей от 0 до 22, функции большей арности выражаются единым интерфейсом `FunctionN` (см. рис. 1.4).

```
1  public interface Function0<out R> : Function<R> {
2      public operator fun invoke(): R
3  }
4  public interface Function1<in P1, out R> : Function<R> {
5      public operator fun invoke(p1: P1): R
6  }
7  public interface Function2<in P1, in P2, out R> : Function<R> {
8      public operator fun invoke(p1: P1, p2: P2): R
9  }
10 //...
11 interface FunctionN<out R> : Function<R>, FunctionBase<R> {
12     operator fun invoke(vararg args: Any?): R
13
14     override val arity: Int
15 }
```

Рис.1.4. Функциональные интерфейсы Kotlin

Интерфейс `FunctionN` не содержит информации о типах аргументов функции, типовой параметр характеризует только возвращаемое значение. Это означает, что в момент компиляции нет возможности проверить корректность типов аргументов вызова, совершенного с помощью FunctionN, что ведет к ошибкам времени выполнения. Наличие механизма для выражения переменного набора типовых параметров может помочь обнаружить подобные ошибки на этапе компиляции. Уточним, что проверке функциональных типов для скомпилированных модулей также препятствует стирание типов в байт-коде Java, о чем будет подробнее рассказано в разделе 3.1.

Рассмотренные примеры показывают, что переменный набор типовых параметров имеет ряд важных практических применений.

1.3. Параметрический полиморфизм в C++

Поддержка параметрического полиморфизма в языке C++ реализована на основе шаблонной кодогенерации. Объявление шаблона может включать типовые и нетиповые параметры, а также вложенные шаблоны в качестве параметров. Кроме того, начиная со стандарта C++11, объявление шаблона может содержать набор параметров (parameter pack). В данной работе именно наборы параметров интересуют нас в первую очередь.

Для примера рассмотрим задание такой шаблонной структуры:

```
1 template<
2     template<typename, typename...> class TT,
3     typename T1,
4     typename... Rest
5 >
6 struct s {};
```

Рис.1.5. Шаблонные параметры в C++

Структура имеет три параметра `TT`, `T1` и `Rest`. `TT` – шаблонный параметр, `T1` – типовой параметр, `Rest` – набор типовых параметров. Отметим, что второй параметр шаблонного параметра `TT` также является набором параметров.

При подстановке данного шаблона, т.е. при создании экземпляра структуры s, в Rest может быть передано произвольное количество шаблонных аргументов, в т.ч. ни одного. Набор шаблонных параметров в объявлении может быть использован в заголовке или теле функции, чтобы обозначить типы одновременно для нескольких значений, как это сделано в следующем примере:

```
1 template<class ...Us> void f(Us... pargs) {}
2 template<class ...Ts> void g(Ts... args) {
3     f(&args...);
4 }
5 g(1, 0.2, "a");
```

Рис.1.6. Применение набора типовых параметров в C++

В примере объявлены две шаблонные функции `f` и `g` с переменным набором аргументов и переменным набором шаблонных параметров. Набор параметров задает тип набора аргументов. Т.е. функции могут принимать произвольное количество аргументов произвольных типов. Также показана распаковка набора аргументов: внутри функции `g` каждый из ее аргументов передается по ссылке в функцию `f`.

Тут стоит отметить, что язык C++ является языком со строгой статической типизацией. Строгая типизация требует, чтобы для каждого объекта программы в момент компиляции был однозначно известен его тип. Статическая типизация означает, что тип объекта остается постоянным на протяжении всего времени жизни этого объекта. С учетом данных ограничений возникает вопрос о способах использования конструкций с произвольным набором параметров произвольных типов так, чтобы мог выводиться тип каждого конкретного аргумента при использовании шаблона. Для того чтобы понять, как озвученные ограничения поддерживаются в языке C++, нужно рассказать, как происходит подстановка шаблонов и как обычно реализуется переход от работы с набором аргументов или параметров к работе с одним элементом набора.

C++ поддерживает перегрузку функций для разного количества и разных типов аргументов и параметрический полиморфизм. Для их обработки используется один и тот же подход, суть которого состоит в генерации компилятором независимого кода для каждого уникального набора параметров. Все полиморфные объекты в процессе компиляции получают уникальные имена на основании их оригинального имени и типов всех фактических параметров по месту использования. Например, у функции параметрами являются ее имя, типы ее аргументов и шаблонные переменные. Алгоритм генерации гарантирует, что новые имена будут уникальными для разных параметров и совпадающими для одинаковых. В результате работы компилятора создаются необходимые экземпляры функций для всех случаев использования в

программе, а выбор одной из этих функций детерминирован за счет однозначного сопоставления по именам. В процессе генерации такого шаблонного кода компилятор выбирает из доступных ему объявлений с нужным исходным именем наиболее подходящее объявление как по аргументам, так и по шаблонным параметрам.

При обработке набора параметров в шаблоне используется следующие правила:

- Все шаблонные аргументы, которым не нашлось одиночного параметра, попадают в набор параметров, если такой имеется в объявлении и его тип подходит для аргумента шаблона;
- Из двух подходящих под вариант использования шаблонных объявлений выбирается то, в котором наименьшее количество приходится на набор параметров. Другими словами, будет выбрано более специфичное объявление.

Приведем пример [2], который показывает, как эти правила могут использоваться на практике.

```
1 template <typename... args> struct count;
2 template <typename T, typename... Args>
3 struct count<T, Args...> {
4     static const int value = 1 + count<Args...>::value;
5 };
6 template <> struct count<> {
7     static const int value = 0;
8 };
```

Рис.1.7. Структура, считающая количество типовых параметров

В данном примере описывается объект, хранящий количество подставленных в него шаблонных переменных. Создается три объявления шаблона с одинаковым именем: базовый шаблон, шаблон, обрабатывающий первый параметр из набора и рекурсивно передающий обработку оставшихся параметров, а также терминирующий шаблон, завершающий рекурсию для пустого набора параметров. Во время компиляции использование шаблона с конкретным набором типовых

параметров превращается в цепочку подстановок рекурсивного шаблона, которая завершается шаблоном без параметров.

Рассмотренные примеры демонстрируют два возможных способа работы с набором типовых параметров в языке C++:

- Передачу всего набора параметров без изменений, либо после применения одинаковых преобразований для всех элементов набора;
- Выделение одного шаблонного параметра из набора за счет механизма спецификации шаблонов и работа с ним как с обычным параметром.

Также стоит отметить и недостатки, которыми обладает реализация полиморфизма в языке C++ посредством шаблонов:

- Генерация внутри компилятора отдельного экземпляра шаблона для каждого уникального случая его использования требует дополнительного времени для компиляции и приводит к увеличению размера исполняемого кода;
- Обнаружение ошибок, вызванных несовпадением типов в шаблонных объектах, откладывается до момента подстановки параметров в шаблон. Из-за этого при разработке библиотек статическая типизация не позволяет обнаружить ошибки в типах, большая ответственность за их выявление ложится на модульное тестирование.

1.4. Параметрический полиморфизм в D

Параметрический полиморфизм языка D [10] отчасти похож на используемый в языке C++, однако имеет ряд существенных отличий. Основу параметрического полиморфизма в D составляют шаблонные объявления, но само понятие шаблона является намного более самодостаточным. В C++ шаблоны не являются сущностью сами по себе,

но в комбинации с другими объявлениями создают параметризированные типы. В D шаблон напоминает пространство имен, все объявления внутри которого разделяют один набор шаблонных параметров.

```
1 template Empty(T) {}
2 alias emptyTemplateInstance = Empty!(int);
```

Рис.1.8. Пустой шаблон

В Листинге 1.8 приведено объявление пустого шаблона и создание экземпляра путем подстановки типовой переменной. Как видно из примера, шаблон может существовать, даже если в его теле не содержится других объявлений.

Использование шаблона становится возможным после создания его экземпляра, т.е. после передачи шаблонных аргументов. В большинстве случаев это должно быть сделано явно. Параметрами шаблона могут быть типы, значения или символы (alias), ограничение задается в объявлении шаблонного параметра, как это показано на рис. 1.9.

```
1 template Foo(T)         { ... }
2 template Foo(int n)     { ... }
3 template Foo(alias sym) { ... }
```

Рис.1.9. Различные типы шаблонных параметров

Последний параметр шаблона может быть превращен в последовательность шаблонных параметров с помощью модификатора «...». В этом случае все шаблонные аргументы, которые не могут быть сопоставлены с базовыми параметрами, будут считаться частью последовательности. Если все элементы последовательности являются типами, такая последовательность может быть использована в объявлении функции для обобщенного задания типов ее аргументов. Сами аргументы при этом выражаются последовательностью значений. Последовательность также может включать комбинацию разных типов шаблонных параметров. Примеры последовательностей приведены на

рис. 1.10. В шаблоне `print` используется последовательность значений, в шаблоне `write` – последовательность типов.

```
template print(args...) {
    void print()
    {
        writeln("args are ", args);
    }
}

template write(Args...) {
    void write(Args args)
    {
        writeln("args are ", args);
    }
}

void main() {
    print!(1,'a',6.8).print();
    write!(int, char, double).write(1, 'a', 6.8);
}
```

Рис.1.10. Виды последовательностей

Помимо передачи целиком, возможна рекурсивная обработка последовательнстей шаблонных параметров. На каждом шаге рекурсии первый шаблонный аргумент последовательности обрабатывается, а остальные попадают в новую последовательность уменьшенной длины. На рис. 1.11 показан пример функции, печетающей переданные аргументы таким способом.

```
template print(T, Args...) {
    void print(T first, Args args) {
        writeln(first);
        static if (args.length)
            print(args);
    }
}

void main() {
    print(1, 'a', 6.8);
}
```

Рис.1.11. Рекурсивная обработка шаблонных параметров

В примере проверка окончания рекурсии производится с помощью оператора `static if (args.length)`. Свойство `length` имеется

у всех последовательностей и известно в момент компиляции программы, поэтому может быть использовано в статическом условии времени компиляции. Помимо вычисления длины, возможно извлечение элементов последовательности по индексу и взятие подпоследовательности от исходной последовательности. Однако последовательности должны быть вычисляемы в момент компиляции и не могут быть изменены в дальнейшем, как при компиляции, так и при выполнении программы.

Язык D также поддерживает более лаконичную запись и вывод параметров шаблонов в отдельных случаях. Если объявление внутри шаблона совпадает с именем самого шаблона, такое обяъявление может быть использовано без указания имени второй раз и будет подразумеваться неявно. В примере (рис. 1.10) можно сократить функцию main с сохранением смысла первоначальных вызовов таким образом:

```
1 void main {
2     print!(1,'a',6.8)();
3     write!(int, char, double)(1, 'a', 6.8);
4 }
```

Рис.1.12. Сокращенное обращение к одноименным свойствам шаблона

Шаблонные функции являются частным случаем шаблона с одноименными объявлениями. Чтобы шаблон считался шаблонной функцией, должны выполняться следующие условия:

- Шаблон должен содержать ровно одно объявление функции;
- Имя функции должно совпадать с именем шаблона.

В случае, если условия выполняются, допускается сокращенная запись для объявления шаблонной функции. Кроме того, если шаблонные аргументы выводятся из аргументов функции, их можно не указывать. Функция `write` (см. рис. 1.10) может быть приведена к виду шаблонной функции, как это показано на рис. 1.13. Список шаб-

лонных параметров в объявлении функции предшествует списку аргументов.

```
1 void write (Args...)(Args args) {
2     writeln("args are ", args);
3 }
4
5 void main() {
6     write(1, 'a', 6.8);
7 }
```

Рис.1.13. Шаблонаая функция write

Для каждого использования шаблона с уникальным набором аргументов в момент компиляции создается отдельный экземпляр. Все применения с одним набором аргументов используют один и тот же экземпляр шаблона. Такая обработка шаблонов имеет все те преимущества и недостатки, что и система шаблонов в C++.

Подводя итог, можно сказать, что шаблоны языка D во многом напоминают шаблоны C++:

- Формирование исполняемого кода шаблона производится после подстановки параметров в месте использования;
- Два основных способа обработки набора (последовательности) шаблонных параметров – передача всего списка целиком и рекурсивная обработка элементов;
- Для работы с набором аргументов используется несколько встроенных операторов (`sizeof...` в C++, `.length` и подпоследовательности в D);
- Обработка разных типов одного и того же параметра реализуются с помощью перегрузок.

Основное отличие переменного набора шаблонных параметров в D от языка C++ – возможность передавать разнородную последовательность аргументов (типов, значений и символов) по одному имени. Во многом это допустимо из-за обязательного указания аргументов шаблона в месте его использования. Разнородная последовательность

обычно обрабатывается более специфичными шаблонами, принимающими только определенный вид параметров. Выбор для каждого отдельного параметра из последоваетльности происходит в момент компиляции по факту использования.

1.5. Параметрический полиморфизм в функциональных языках программирования

Исследование параметрического полиморфизма является актуальной задачей не только для императивных, но и для функциональных языков программирования [5]. В том числе внимание уделяется и проблеме выражения переменной арности функций в сочетании с параметрическим полиморфизмом [8, 14].

В статье [14] предлагается подход к созданию каркаса для определения функций, полиморфных по типам и количеству аргументов, на языке Agda 2. Важной особенностью языка Agda 2 является поддержка зависимых типов, без которых данный метод не может существовать.

Зависимые типы, как следует из названия, позволяют выражать типы, зависящие от значений других типов. Например, тип для списка заданной длины зависит от двух параметров: типа входящих элементов и длины списка. Длина является параметром-значением, фиксированным натуральным числом, делающим тип списка зависимым типом. Более формально, зависимый тип определяется функцией, которая разбивает исходный тип (например, список элементов заданного типа) на множество результирующих типов (множество типов списков разной длины с заданным типом элементов) по значениям (значение – натуральное число).

Суть подхода [14] состоит в использовании функции-генератора для создания прикладной функции, полиморфной произвольному числу типов. Тип генератора приведен на рис. 1.14.

```
1 ngen : {n : N} {b : Vec Set (suc n) -> Set} {k : Kind}
2 -> (t : Ty k)
3 -> (ce : TyConstEnv n b)
4 -> b  k  (repeat [t])
```

Рис.1.14. Функция-генератор

Генератор принимает следующие аргументы:

- Арность функции `n`
- Функцию `b`, отображающую вектор типов-параметров длины `n` (зависимый тип от `n`) в тип прикладной функции, зависящий от этих параметров
- Константное окружение `ce` – функцию из типов констант-аргументов конечной функции в типы-индексы, принимаемые функцией `b`
- Род и тип аргумента прикладной функции, передаваемые из объявления прикладной функции

Генератор реализует логику создания абстрактного типа для функции переменной арности с полиморфными типами, который бы зависел от конкретных типов переданных аргументов. Использование генератора предполагает определение функций `b` и `ce` в соответствии с логикой прикладной функции. Функция b выражает зависимость типа создаваемой функции от всего списка типовых параметров. Пример (см. рис. 1.15) показывает, как функция `b` может быть определена для создания бинарной функции `map` на основе генератора. Ее аргумент – вектор из двух типовых параметров A и B, а результат – тип функции $A \to B$. Уточним, что функция работает не со значениями типов, а с самими типами.

```
1 Map : Vec Set 2 -> Set
2 Map (A :: B :: [ ]) = A -> B
```

Рис.1.15. Объявление функции b для map

Константное окружение `ce` определяется для конкретной функ-

ции b. Для приведенной функции Map, задача ce – на основе аргументов создать конкретный тип функции $A \to B$. Полное определение функции ce рассматривает реализацию для всех типов аргумента, включая поддерживаемые языком Agda типы-произведения и типы-суммы. Для объяснения сути подхода ограничимся рассмотрением только константных типов аргументов. Для них в контексте функции map ожидаемым типом будет тип функции из вектора типов значений до применения функции map в вектор типов значений после ее применения. Получить его позволяет композиция стандартных функций repeat, повторяющей свой аргумент по числу аргументов, с которыми вызывается прикладная функция, и оператора получения типа по значению аргумента. Число аргументов прикладной функции на этапе объявления неизвестно, но может быть выведено в момент компиляции. Объявления неявных, выводимых компилятором аргументов поддерживаются синтаксисом языка – они объявляются в фигурных скобках.

Определив таким образом функции b и ce, можно создать объявление функции map, передав константное окружение в функцию-генератор. Полученная в результате функция является параметрической по типам аргументов, но не по их количеству, поскольку в примере константное окружение никак не использует переданную ей в качестве аргумента арность. Использование этого аргумента позволяет обобщить логику map на произвольное количество списков аргументов, превращая тип из $A \to B$ в тип $A \to B \to C \to ...$ выводимой компилятором длины по аналогии с тем, как это было сделано для длины вектора аргументов. В Листинге 1.16 показано рекурсивное объявление функции b для функции map переменной арности.

```
1 arrTy : {n : N} -> Vec Set (suc n) -> Set
2 arrTy {0} (A :: [ ]) = A
3 arrTy {suc n} (A :: As) = A -> arrTy As
```

Рис.1.16. Объявление функции b для map переменной арности

Теперь вместо обработки двух типов функция принимает вектор типов и возвращает тип каррированной функции, эквивалентной функции от списка аргументов.

Зависимые типы имеют большую выразительную силу, но значительно усложняют систему типов. Из-за этого их поддержка в языках программирования распространена не так широко. Большинство языков, использующих зависимые типы, являются научными. Среди них полностью императивным является только язык Xanadu [15]. Главным обоснованием для добавления зависимых типов в этот язык, по мнению его создателей, является контроль индексирования на уровне системы типов и выражение дополнительных свойств, связанных со значениями аргументов. Язык позволяет избежать ошибок, вызванных выходом на границы выделенной памяти, и гарантировать большую надежность программ. Однако синтаксис языка не позволяет использовать зависимые типы для выражения переменной арности.

Разработчики Typed Scheme, осознавая сложность реализации зависимых типов, выбрали альтернативный подход для поддержки переменного набора типовых параметров в своей системе типов [8]. Typed Scheme (в настоящее время носит название Typed Racket) – проект по созданию статически типизированного диалекта языка PLT Scheme, позволяющего типизировать существующий код, написанный на PLT Scheme. В результате анализа кодовой базы авторы пришли к выводу, что функции переменной арности достаточно широко применяются на практике, поэтому специальный синтаксис для объявления таких функций является востребованным.

В статье выделяется два основных подтипа для переменного набора аргументов: однородный и разнородный. Все параметры однородного набора имеют один и тот же тип, типы разнородного набора параметров могут отличаться. Для выражения однородного набора используется тип, помеченный с помощью модификатора «*». Такой маркер может иметь только тип последнего аргумента функции. В

свою очередь разнородный набор параметров похож по синтаксису на переменные шаблонные параметры в C++ и D. Для создания разнородного набора аргументов с помощью маркера ... обозначается типовая переменная переменной длины, после чего с ее помощью может быть выражен переменный набор типов. Этот набор может быть использован для обозначения типа аргумента.

```
1 (: + (Integer* -> Integer))
2
3 (: map
4 (forall (R A B... )
5 ((A B ...b -> R) (Listof A) (Listof B)...b -> (Listof R))))
```

Рис.1.17. Объявление типов функций + и map

На рис. 1.17 приведены примеры типов функций, принимающих однородный и разнородный набор аргументов. Функция + складывает произвольное количество целых чисел, все ее аргументы имеют одинаковый тип. map принимает функцию от одного аргумента или более и соответствующее число списков аргументов разных типов и возвращает результат последовательного применения функции на аргументах из списков. i-ый список соответствует i-му аргументу функции.

Авторы отмечают, что их система типов имеет ограничения и не позволяет типизировать все функции переменной арности. Примером являются функции обработки списков, такие как `sort` и `filter`. В их поведении с точки зрения типов есть существенное отличие от функции `map`: они могут менять размер и позицию аргументов принимаемого списка. Из-за этого тип аргумента-списка обязан быть однородным. Компилятор Typed Scheme допускает преобразование из разнородного списка аргументов в однородный, но не наоборот. Соответственно, результат применения функций, приводящих разнородный список аргументов к однородному, не может быть использован в качестве аргумента другой функции, оперирующей разнородным набором аргументов.

2. ЦЕЛИ И ЗАДАЧИ РАБОТЫ

2.1. Цель работы

Цель данной работы состоит в анализе возможностей по добавлению поддержки переменного набора типовых параметров в язык Kotlin, а также в разработке версии компилятора, поддерживающего расширенную версию синтаксиса.

2.2. Задачи работы

Для достижения цели работы требуется выполнить следующий набор задач:

- Проанализировать существующие подходы для выражения переменного набора типовых параметров в языках программирования, оценить возможности по их адаптации для Kotlin;
- Разработать синтаксис, допускающий переменный набор типовых параметров;
- Внести в компоненты компилятора изменения, требуемые для поддержки обновленного синтаксиса;
- Провести функциональное тестирование прототипа;
- Провести регрессионное тестирование, оценить совместимость с предыдущими версиями компилятора;
- Дать оценку возможностей и ограничений разработанного синтаксиса и реализующей его версии компилятора.

3. РАЗРАБОТКА СИНТАКСИСА ПЕРЕМЕННОГО НАБОРА ТИПОВЫХ ПАРАМЕТРОВ

3.1. Особенности языка Kotlin

Kotlin – статически типизированный язык программирования общего назначения. В настоящее время Kotlin поддерживает компиляцию в формат виртуальной машины Java (JVM), в JavaScript и в нативный машинный код, однако первоначально язык работал только с JVM. Совместимость Kotlin и Java позиционируется как одно из важных достоинств языка и во многом определяет его дизайн и ограничения. Под совместимостью понимается возможность импортировать код, написанный на Java, в программах на Kotlin и наоборот. Другие языки, использующие JVM, такие как Scala, во многих случаях не поддерживают обратную совместимость с Java, и библиотеки на них часто не могут быть импортированы из кода на Java. Отметим, что использование в коде отдельных возможностей языка Kotlin также препятствует его импорту из Java, но на данный момент такие специфичные конструкции являются по большей мере исключением из общего правила.

JVM – виртуальная машина, выполняющя код, представленный в формате .class-файлов, также называемых Java байт-кодом. Промежуточное представление программы в виде байт-кода позволяет абстрагироваться от конкретной аппаратной архитектуры системы, на которой программа будет выполняться. Запуск кода .class-файлов, происходит с помощью среды выполнения Java (JRE) и состоит из нескольких этапов [12]:

- загрузчик проверяет корректность байт-кода и загружает его в рабочую память виртуальной машины;
- обработчик команд интерпретирует и выполняет байт-код;
- часто исполняемые участки кода компилируется JIT-компилято-

ром в машинный код для повышения производительности.

Язык Java оказал существенное влияние на то, какой вид принял параметрический полиморфизм в Kotlin из-за явления, называемого стиранием типов (type erasure) в байт-коде. До версии Java 5.0 в языке не существовало синтаксиса для объявления типовых переменных. В момент их добавления встал вопрос обратной совместимости с предыдущими версиями языка [4]. Например, новая версия стандартной библиотеки переопределяла интерфейсы коллекций, такие как `java.util.List`, делая их параметризированными относительно типа элементов. В новой версии Java все старые использования `List` без параметров стали неявно параметризированными с параметром `Object` – базовым классом иерархии типов Java. Сложность вызывал вопрос, что делать с типовыми параметрами в момент выполнения программы. Очевидно, что в старых версиях скомпилированного байт-кода не содержится информации о типовых параметрах. Это означает, что попытка получить информацию о типе параметра в момент выполнения программы невозможна при использовании кода ранних версий Java. В целях сохранения совместимости было решено отказаться от сохранения типов параметров в байт-коде новых версий и сделать их неовеществляемыми (non-reifiable). Из-за этого динамические обращения к типу в момент выполнения, например, с использованием оператора `instanceof`, невозможны для типовых параметров.

Kotlin поддерживает параметрический полиморфизм для классов, интерфейсов и функций. Для поддержания совместимости с Java тип типовых параметров стирается в момент генерации байт-кода и недоступен во время выполнения программы. Примеры полиморфных объявлений приведены в листинге (см. рис. 3.1). Типовые параметры задаются в треугольных скобках.

Из правила, запрещающего обращение к типу типового праметра в момент исполнения, существует исключение. Kotlin позволяет создавать функции с модификатором `inline`, при наличии которого

```
class GenericClass <T> (val field: T)
interface GenericInterface <T> {
    fun doStuff(argument: T) {
        // ...
    }
}
fun <T> genericFunction(argument: T) {
    // ...
}
```

Рис.3.1. Примеры полиморфных объявлений в Kotlin

все вызовы данной функции заменяются на подстановку ее тела. Типовые параметры таких функций становятся овеществленными, если помечены модификатором `reified`.

Kotlin также позволяет объявлять функции с переменным набором аргументов. Аргумент, обозначенный ключевым словом vararg, принимает массив значений одинакового типа при вызове. Тип значений может быть задан типовым параметром, что позволяет создавать функции переменной арности с однородным набором аргументов. Пример такой функции показан на рис. 3.2.

```
fun <T> homogeneous(vararg arguments: T) {
    // ...
}
```

Рис.3.2. Полиморфная функция с однородным списком аргументов

В контесте разговора о типовых параметрах стоит также рассказать о понятии вариантности типового параметра и типовых проекциях. Отношение наследования между двумя типами не сохраняется, если они становятся типовыми аргументами другого типа. Вариантность позволяет устанавливать подобные отношения, накладывая необходимые ограничения на использование типового параметра. В качестве примера рассмотрим интерфейс `List`, показанный на рис. 3.3.

Тип `List<Int>` не будет подтипом `List<Number>`, несмотря на то, что тип `Int` наследуется от типа `Number`. Данное ограничение необходимо, поскольку иначе использование `List` перестает быть типобез-

```
1 interface List<T> {
2     fun add(item: T)
3     fun get(index: Int): T
4     //...
5 }
```

Рис.3.3. Интерфейс List

опасным. Пример такого использования показан на рис. 3.4.

```
1 val intList: List<Int> = //...
2 val numberList: List<Number> = intList
3 numberList.add(5.6)
4 val int: Int = intList.get(0)
```

Рис.3.4. Нетипобезопасное использование List

В примере созданный список целых чисел сохраняется по ссылке как список произвольных чисел, что допустимо, если считать `List<Int>` подтипом `List<Number>`. После этого через интерфейс `List<Number>` в список добавляется элемент с типом Double, а затем извлекается через интерфейс `List<Int>` и интерпретируется как целое число. Подобный код приведет к ошибке времени выполнения, т.к. `Double` не может быть преобразован к `Int`.

Интерфейс `List` является инвариантным, поскольку его функции используют тип типового аргумента как во входной позиции (в качестве аргумента функции `add`), так и в выходной позиции (возвращаемое значение функции `get`). Тем не менее, для многих типов типовая переменная используется только в одной из этих двух позиций. В этом случае можно сделать интерфейс типа более гибким, если объявить типовую переменную с необходимой вариантностью. Если использования типовой переменной ограничены только выходными позициями, она является ковариантной и может быть обозначена в Kotlin модификатором `out`. Например, интерфейс `kotlin.collections.List` является ковариантным, поскольку содержит только операции, связанные с чтением элементов списка, а сам список неизменяем после создания. Аналогично, контрвариантные типовые переменные допускают

использование только во входных позициях и обозначаются модификатором in.

Подставленное значение типового параметра в месте использования образует проекцию типа – сочетание типа и вариантности. Вариантность проекции определяется сочетанием вариантности в определении типа и модификатором вариантности типового аргумента. Фактически, только итоговая вариантность проекции влияет на допустимые операции над объектом. Пример на рис. 3.5 показывает, как небезопасное использование инвариантного интерфейса `List` может быть исправлено с помощью модификатора вариантности. В новой версии кода `numberList` не допускает операцию записи из-за указанной вариантности, но позволяет оперировать элементами списка как объектами с типом `Number`.

```
1 val intList: List<Int> = // ...
2 val numberList: List<out Number> = intList
3 val int: Int = intList.get(0)
4 val number: Number = numberList.get(0)
5 // numberList.add(5.6) // compilation error
```

Рис.3.5. Безопасное использование List

3.2. Дизайн синтаксиса

В качестве первого шага для создания синтаксиса было рассмотрено то, как способ выражения переменных шаблонных параметров в языке C++ может быть перенесен на Kotlin с учетом различий между языками. Отстутсвие шаблонной кодогенерации является главной причиной, по которой существующий синтаксис C++ не может быть использован «как есть». Типовые переменные в Kotlin, хотя и выполняют схожую смысловую роль с шаблонными параметрами, обрабатываются компилятором принципиально иначе. Чтобы понять ограничения, которые из-за этого возникают, обратимся к рассмотренному ранее примеру объявления и использования функции с переменным

набором шаблонных параметров на C++ (см. рис. 1.6).

Для того чтобы определить корректность вызова `g()`, в момент компиляции должна быть возможность проверить, что аргументы `args` с типами `Ts...` совпадают по типам с параметрами функции `f`. Это возможно в языке C++ благодаря генерации исполняемого кода для функции `f` уже после подстановки аргументов в функцию `g`. Проверка типов сводится к поиску объявления функции `f` с подходящими типами аргументов. Отсутствие объявления `f` с необходимыми типами аргументов означает ошибку использования функции `g`.

Попробуем перенести пример на язык Kotlin с сохранением исходного смысла. Для начала создадим синтаксически корректное определение функции g на существующей версии Kotlin. Язык поддерживает аргументы функции переменного размера, однако их тип обязан быть одинаковым.

```
1 fun <Ts> f (vararg args: Ts) { /*...*/ }
2 fun <Ts> g (vararg args: Ts) {
3   f(*args)
4 }
5
6 g(1, 0.2, "a")
```

Рис.3.6. Перенос с C++ на Kotlin, версия 1

Объявление функции `g` (рис. 3.6) корректно синтаксически, но семантически отличается от исходного примера. В теле функции `g` уникальность типов аргументов теряется, т.к. список аргументов однороден. Теперь представим, что существует возможность объявить набор разнородных типовых параметров по аналогии с набором шаблонных параметров.

```
1 fun <Ts...> f (vararg args: Ts) { /*...*/ }
2 fun <Ts...> g (vararg args: Ts) {
3   f(*args)
4 }
5
6 g(1, 0.2, "a")
```

Рис.3.7. Перенос с C++ на Kotlin, версия 2

Пример в листинге на рис. 3.7 показывает, как разнородный набор параметров может быть объявлен. Но объявление функции с разнородным переменным набором аргументов будет не полезнее функции с однородным набором, если у компилятора нет возможности статически проверить корректность вызова на основании типов аргументов. В C++ с помощью специализации шаблона и рекурсивной обработки шаблонных параметров аргументы функции `f` могут быть использованы как `int`, `double`, `std::string`. При этом компилятор может гарантировать, что типы корректны, т.к. поиск подходящего экземпляра шаблона основывается на типах переданных аргументов, и происходит в момент применения, а не объявления g.

В Kotlin типовые параметры не обрабатываются с помощью шаблонной генерации. Для всех вызовов решение о том, какие именно объявления будут использованы происходит на основе имени и известной информации о типах аргументов. Из-за этого компилятор не может отложить проверку типов до момента вызова, когда вся информация о типовых параметрах будет известна. Это делает невозможной рекурсивную обработку типовых параметров, являющуюся основным способом работы с шаблонными параметрами в C++. Пример, поясняющий проблему, приведен в листинге (см. рис. 3.8).

```
1  fun <T, Ts...> process(first: T, vararg rest: Ts) {
2      processSingle(first)
3      process(*rest)
4  }
5
6  fun process() {}
7
8  fun processSingle(argument: Type1) { /* ... */ }
9  fun processSingle(argument: Type2) { /* ... */ }
10 // ... other supported declarations of processSingle
```

Рис.3.8. Проблема рекурсивной обработки параметров в Kotlin

Идея примера – задать функцию, которая бы выделяла и обрабатывала первый аргумент, а остальные передавала в рекурсивный вызов. Проблема в данном примере состоит в том, что для вызова

`processSingle(first)` компилятору требуется выбрать, какая именно версия функции используется в данном месте. Но по смыслу вызываемая функция должна быть разной для разных типов аргументов, выбрать одну из них принципиально невозможно. Объединение функций `processSingle` в одно объявление (см. рис. 3.9) с анализом типа аргумента внутри тела функции, на первый взгляд, позволяет решить проблему. Однако такой подход сводит на нет смысл переменного набора типовых параметров, поскольку польза от информации о типе аргумента теряется в момент его передачи в функцию, работающую с аргументом произвольного типа.

```
1  fun <T, Ts...> process(first: T, vararg rest: Ts) {
2      processSingle(first)
3      process(*rest)
4  }
5
6  fun process() {}
7
8  fun processSingle(argument: Any?) {
9     when (argument) {
10        is Type1 -> { /* ... */ }
11        is Type2 -> { /* ... */ }
12        // ... other supported types of processSingle
13    }
14 }
```

Рис.3.9. Проверка типа аргумента внутри функции

Ограничения языка Kotlin, препятствующие рекурсивной обработке переменного набора типовых параметров, натолкнули на идею явного обращения к отдельным типам из набора. Идея индексирования по позиции типового параметра напоминает способ обращения к элементам последовательности шаблонных параметров в языке D и подход к созданию полиморфных функций переменной арности с помощью зависимых типов на языке Agda.

Смысл подхода состоит в том, чтобы трактовать типовую переменную, задающую переменный набор типовых параметров, как список типов, индексируемый по позиции. Индексирование дает возможность обобщить логику, использующую типы из набора типовых па-

раметров, не привязываясь к их позиции. Именно в этом и состоит основная идея полиморфных функций переменной арности: типовые параметры по отдельности разнородны, но тип, возникающий при комбинации позиции аргумента с его значением, становится однородным. В дальнейшем будем называть данный однородный тип индексированным типом. Индексированный тип можно рассматривать как зависимый тип с ограниченным набором ситуаций, в которых он может быть объявлен. Назначение нового синтаксиса – предоставить понятный и достаточно удобный механизм для обозначения и использования индексированного типа в полиморфных объявлениях.

На значения индекса должны накладываться дополнительные ограничения, поскольку обращение по нему требуется для вывода типов в процессе компиляции программы. Это означает, что значения индекса должны быть константами времени компиляции. Кроме того, индекс должен попадать в диапазон допустимых значений. Границы диапазона задаются фактическим количеством типовых аргументов полиморфного объекта в месте использования.

Kotlin не поддерживает зависимые типы, а обычные типы не являются объектами первого рода и не могут использоваться в качестве значений без средств reflection API. Поэтому при создании синтаксиса для индексированных типов стоял вопрос компромисса между выразительностью и трудоемкостью реализации. В результате был сформулирован следующий набор правил для создания объявлений с переменным набором параметров и создания индексированных типов:

- Набор типовых параметров в объявлении обозначается модификатором «...» после имени типовой переменной;
- Объявление может иметь не более одного набора типовых параметров;
- Набор типовых параметров обязан находиться в последней позиции;

- Если переменная, задающая набор типовых параметров, используется в качестве типа для переменного набора аргументов, такой набор аргументов становится разнородным. Типы и количество аргументов-значений должны совпадать с типами и количеством типовых аргументов;
- Индексированный тип выражается типовой переменной функции, один из аргументов которой является индексом типа;
- Индекс типа представляется в виде в виде аргумента с типом `@TypeIndex kotlin.Int` ;
- Аннотация `@TypeIndex` принимает два аргумента: набор типовых параметров и типовую переменную, обозначающую индексированный тип. Аргументы передаются в виде объектов класса `kotlin.reflection.KClass`;
- Явная декларация типа может содержать * в качестве проекции для типового параметра переменного размера. Такая проекция интерпретируется как произвольное количество произвольных типов. Модификаторы вариантности in и out не поддерживаются текущей версией синтаксиса;
- Функции, использующие индексированные типы, должны вызываться только на объектах с переменным набором типовых параметров.

В качестве примера использования введенного синтаксиса рассмотрим реализацию кортежа в виде класса с переменным набором типовых параметров. Код приведен в листинге на рис. 3.10.

В примере типовая переменная `Ts` задает набор типовых параметров. Аргумент конструктора `values` имеет модификатор `vararg` и тип `Ts`, что делает данный набор аргументов разнородным. Функция `get` использует синтаксис создания индексированного типа. Индекс `ix` имеет тип `Int` и обозначен аннотацией `@TypeIndex`, которая связывает индекс с набором типовых параметров `Ts` и типовой переменной `T`, делая тип последней индексированным типом. Функция `test` показы-

```
1  class Tuple<Ts...> (vararg values: Ts) {
2      private val _values: Array<Any?> = arrayOf(*values)
3
4      fun <T> get(
5          ix: @TypeIndex(Ts::class, T::class) Int
6      ): T = _values[ix] as T
7  }
8
9  fun test() {
10     val tuple: Tuple<Int, String> = Tuple(5, "str")
11     val int: Int = tuple.get(0)
12     val string: String = tuple.get(1)
13
14     /*
15     val index = 1
16     val error1 = tuple.get(2)        // compilation error
17     val error2 = tuple.get(index)    // compilation error
18     */
19 }
```

Рис.3.10. Пример класса кортежа

вает, как класс `Tuple` может быть использован. В функции создается экземпляр класса, хранящий значения типов `Int` и `String`. Поскольку функция `get` возвращает индексированный тип, вызовы `get(0)` и `get(1)` будут иметь тип типового параметра `tuple` по соответствующему индексу, т.е. `Int` и `String`. Благодаря этому не требуется явное приведение типов: компилятор может гарантировать их соответствие. Вызов функции `get` на экземпляре `tuple` с индексами, отличными от 0 и 1 приведет к ошибке компиляции из-за выхода за границы списка типовых аргументов. Передача переменной вместо литерала также недопустима, поскольку значение переменной в общем случае не является константой времени компиляции.

4. РАЗРАБОТКА ПРОТОТИПА КОМПИЛЯТОРА

4.1. Архитектура компилятора

Процесс компиляции программы на Kotlin включает три основных этапа:

- Постороение дерева разбора. В Kotlin используется дополненное представление в виде интерфейса структуры программы (PSI), предоставляемого платформой IntelliJ [6];
- Статический анализ дерева разбора;
- Генерацию кода.

Синтаксический разбор и статический анализ происходят во фронтенде компилятора, вне зависимости от конечной платформы, после чего результаты анализа передаются для кодогенерации на целевой бекенд. Для JVM на бекенде происходит генерация байт-кода, для веба – JavaScript кода, для Kotlin Native – бинарного машинного кода.

Процесс построения синтаксического дерева начинается с работы лексического анализатора. Он выделяет в тексте программы лексемы в соответствии с синтаксисом языка. Лексический анализатор компилятора Kotlin создается с помощью генератора JFlex [3]. Требуемое поведение анализатора задается спецификацией в формате .flex. В ней содержится набор регулярных выражений и описание того, какие лексемы должны создаваться при их обнаружении.

Поток лексем, создаваемый лексическим анализатором, затем передается на обработку в парсер. Его задача – интерпретировать смысл лексем с учетом контекста, поскольку одни и те же лексемы могут выражать разный смысл в зависимости от позиции в программе. Например, знак «<» может выражать как сравнение, так и начало списка типовых параметров. Результатом работы парсера является дерево разбора.

Представление программы в форме дерева разбора упрощает ее последующий анализ, в ходе которого создается вся необходимая для генерации исполняемого кода информация. На данном этапе создаются объекты, описывающие логику программы в высокоуровневом виде. Этой информации должно быть достаточно, чтобы на ее основе можно было однозначно сгенерировать исполняемый код, и при этом

она должна быть до необходимой степени абстрактной, чтобы поддержать генерацию кода на разных бекендах. Характерной для анализа сущностью является дескриптор – объект, выражающий высокоуровневое понятие языка. Дескрипторы описывают функции, классы, параметры, переменные и т.д. и заполняются по мере анализа программы. Для хранения результатов анализа используется структура данных, называемая контекстом связывания. Она представляет из себя двухуровневый ассоциативный массив. Ключи первого уровня разделяют структуру на смысловые слои. Ключи второго уровня зависят от назначения слоя, часто это лексемы дерева разбора определенного вида. Например, в слое может храниться сопоставление лексемы объявления класса с описывающим его дескриптором или информация о вызовах функций со всеми необходимыми контекстными данными.

На этапе кодогенерации собранная в результате анализа информация из контекста связывания используется для создания исполняемого кода. В рамках данной работы обработка переменного набора типовых параметров заканчиваются на этапе анализа и не затрагивают процесс генерации кода.

На каждом этапе комиляции в программе могут быть обнаружены ошибки. На этапе разбора это синтаксические ошибки, на этапе анализа – семантические. Обнаружение ошибки не является исключительной ситуацией и не приводит к мгновенному окончанию компиляции. Типовой способ обработки ошибок – использование объектов-заглушек вместо настоящих объектов, которые не могут быть созданы. Впоследствии обнаруженные ошибки отображаются на элементы PSI, соответствующие участку кода с ошибкой.

4.2. Поддержка типизации в компиляторе Kotlin

С точки зрения компилятора, тип, описывающий объект программы, создается следующим набором атрибутов:

- Конструктором типа – его основной идентифицирующей частью. Конструктор содержит информацию о супертипах данного типа и всех типовых параметрах. В нем же содержится информация о том, могут ли у типа быть наследующие его подтипы;
- Типовыми аргументами – значениями подставленных типовых параметров;
- Флагом, помечающим способность объекта с данным типом иметь значение null.

Одна из важных особенностей Kotlin – поддержка вывода типов. Язык позволяет опускать явные декларации типовых аргументов и/или типа, если они могут быть однозначно определены компилятором на основе имеющейся информации. К примеру, объявление переменной может не иметь явной декларации типа, если может быть вычислен тип выражения, использующегося для инициализации переменной.

Вывод типов происходит в ходе решения компилятором системы уравнений относительно типов. Каждое уравнение накладывает ограничение на то, как типы должны соотноситься друг с другом, т.е. какой из двух типов должен быть подтипом другого. Запись $A <: B$ означает, что тип A является подтипом типа B. Отношение равенства двух типов может быть задано в системе уравнений двумя неравенствами для одной и той же пары типов с противоположными знаками. В дальнейшем для краткости оно будет обозначаться знаком равенства. В процессе составления системы уравнений компилятор фиксирует все явно заданные и выведенные ранее типы, неизвестные типы становятся переменными, относительно которых решается система.

Чтобы создать тип объекта, необходимо определить его конструктор, после чего вычислить полный набор типовых параметров и сопоставить им типовые аргументы. Полный набор типовых параметров может быть больше, чем набор типовых параметров объявления в случае вложенных объявлений. Так, если внутри класса объявлен

внутренний класс, тип объектов внутреннего класса будет содержать аргументы для всех типовых параметров обоих классов.

```
1 class Outer<T1> {
2     inner class Inner<T2>
3 }
4
5 val outer = Outer<OuterTypeArgumentType>()
6 val inner = outer.Inner<InnerTypeArgumentType>()
```

Рис.4.1. Вложенный класс

В примере тип переменной `inner` задается ее конструктором `Inner`, двумя типовыми аргументами (`InnerTypeArgumentType`, `OuterTypeArgumentType`) и признаком, что переменная не может иметь значение null.

4.3. Изменения в синтаксическом разборе

Изменения компилятора для поддержки нового синтаксиса затронули два первых этапа его работы: синтаксический разбор и статический анализ.

На этапе синтаксического разбора добавился поиск модификатора «...»:

- В спецификацию, задаваемую .flex-файлом, добавилось регулярное выражение для последовательности «...»;
- В функцию разбора типового параметра добавился шаг проверки наличия модификатора после имени параметра.

Изменения спецификации оказались минимальными и заключались в исправлении имени лексемы, поскольку модификатор уже был зарезервирован для потенциального использования в синтаксисе.

Шаг поиска модификатора затронул процесс построения дерева разбора. PSI строится из потока лексем последовательной проверкой возможных продолжений для текущего состояния парсера. Для этого парсер устанавливает маркеры, которые используются как границы

узла, если узел может быть сформирован из потока лексем, либо служат точкой возврата в противоположном случае. Парсер перебирает допустимые варианты продолжения до тех пор, пока не обнаружит противоречие в последовательности – невалидную, либо недостающую лексему. Тогда ошибочная последовательность лексем помечается в виде специального узла дерева, и разбор продолжается с последней корректной точки.

Узел PSI, обозначающий типовой параметр, включает обязательное имя параметра и опциональный набор предшествующих модификаторов. В новом синтаксисе модификатор «...» может появляться только после имени типового параметра. Прочие лексемы, не заканчивающие разбор типового параметра, в данной позиции недопустимы. Поэтому проверка наличия модификатора производится в момент после обнаружения имени типового параметра до завершения разбора типового параметра. При обнаружении необходимой лексемы в последовательности, она добавляется дочерним узлом к узлу типового параметра.

Остальные модификации синтаксиса выражаются с помощью аннотаций, и потому не требуют дополнительных действий при построении PSI.

4.4. Изменения в статическом анализе

Вслед за синтаксическим разбором выполняется статический анализ программы. Наибольшая часть функций для поддержки нового синтаксиса реализуется на этом этапе.

В разделе 3.2 было сказано, что трудность статического вывода типов объявлений с переменным набором типовых параметров возникает из-за необходимости иметь всю информацию о типовых аргументах в момент компиляции объявления, когда вызов с конкретными типовыми аргументами еще не совершен. Явное обращение к индек-

сированному типу через типовой параметр позволяет отложить обработку типов до момента совершения вызова. Задачей компилятора в этом случае становится вывод индексированного типа на основании индекса и типовых аргументов объекта по месту вызова. Проверка типов разнородного переменного набора аргументов также происходит при обработке вызова, а не объявления.

Соответственно, на этапе анализа появляются дополнительные задачи:

- Проверка объявлений с переменным набором типовых параметров на корректность типовой переменной;
- Вывод и сохранение типов типовых аргументов;
- Проверка разнородного набора аргументов переменной длины;
- Проверка корректности вызова функцкции с индексированным типом.

Рассмотрим их детальнее.

4.4.1. Проверка типового параметра

Типовые параметры описываются в компиляторе отдельным видом дескрипторов. Возможность выражать переменный набор типовых параметров стала обозначаться в нем дополнительным логическим флагом. Флаг устанавливается в момент создания дескирптора, если в PSI у узла типового параметра имеется дочерний узел – модификатор «...». Вся дальнейшая обработка, связанная с анализом переменного набора типовых параметров, использует флаг в дескрипторе типового параметра и не обращается к PSI напрямую.

Обработка ошибок на этапе анализа обычно происходит одним из двух способов:

- Непосредственно в ходе анализа. Для этапа анализа заводится временное хранилище ошибок, и после завершения всех шагов

этапа найденные ошибки переносятся из временного хранилища в глобальный контекст связывания;

- Отдельным шагом после завершения этапа анализа. Проверки выражаются в виде объектов, реализующих общий интерфейс. Напрмер, интерфейс `DeclarationChecker` реализуется всеми проверками объявлений. Интерфейс определяется типом объектов, на который направлены проверки. В частности, такие наборы проверок существуют для объявлений и для вызовов. Внутри проверки имеется доступ к анализируемому объекту и контекстной информации.

В соответствии с новым синтаксисом только последний типовой параметр объявления может быть отмечен модификатором «...». Нарушение этого правила должно проверяться, но проверка может быть отложена до момента, когда дескриптор объявления создан окончательно. Это возможно, поскольку модификации, связанные с обработкой переменного набора типовых параметров, анализируют только последний параметр объявления. Поэтому проверка ошибок выполняется на отдельном шаге проверок объявлений, после создания дескриптора.

В добавленной проверке просматривается список дескрипторов типовых параметров, содержащихся в дескрипторе объявления. Учитываются только те объявления, которые в принципе могут содержать типовые параметры. Если какой-либо из дескрипторов типового параметра, кроме последнего, помечен флагом переменной размерности, диагностируется ошибка.

4.4.2. Вывод и хранение типовых аргументов

Добавление в синтаксис переменного набора типовых аргументов меняет принцип их сопоставления с типовыми параметрами. Связь один параметр – один аргумент меняется на отображение один-ко-мно-

гим. Это изменение важно с точки зрения обратной совместимости.

Для сохранения совместимости используется двойственное представление типовых аргументов для параметра переменной размерности. Первое представление – в виде одного общего типового аргумента, второе представление – в виде списка типовых аргументов. Список – фактический источник данных для работы с переменным набором типовых аргуменов, а общий аргумент нужен для сопряжения со старой системой типов. Благодаря этому задачи разделяются между двумя представлениями, их реализация становится проще. Две формы типового аргумента объединяются друг с другом с использованием аннотированных типов.

Аннотации в программах на Kotlin используются для добавления метаинформации. Они могут служить разным целям, начиная от документирования и заканчивая изменением процесса компиляции. В случае с переменным набором типовых параметров аннотация становится местом хранения информации о типах типовых аргументов. Внутри компилятора аннотированный тип представляется объектом `AnnotatedSimpleType`, состоящим из комбинации типа-делегата и набора дескрипторов аннотаций. Делегат отвечает за базовое представление типа, а аннотации содержат дополнительную информацию.

Компилятор имеет возможность создавать дополнительные дескрипторы, не связанные с явно выраженными в коде объектами программы. С помощью подобного механизма поддерживаются вызовы по конвенции, такие, как операторы неравенства или разложение в набор переменных. При использовании конвенций вызов операторной функции с необходимым именем генерируется в компиляторе искусственно. В случае с переменным набором типовых параметров информация о типовых аргументах сохраняется в синтетическую аннотацию `@TypeArguments`, которая затем добавялется к типу объекта. Аргументы `@TypeArguments` – массив типов подставленных типовых параметров, упакованных в `kotlin.reflect.KClass`.

Создание аннотированного типа для хранения типовых аргументов требуется в двух случаях: для явных объявлений типа и для вызовов конструктора типа с переменным набором типовых параметров в объявлении. Эти случаи затрагивают анализ разных позиций: в первом случае – позицию типа, во втором – позицию выражения. Для пояснения приведем пример.

```
1 val a: Collection<Int> = listOf(1,2,3)
2 val b = listOf(1,2,3)
```

Рис.4.2. Позиция типа и позиция выражения

В листинге на рис. 4.2 приведены два варианта создания переменной. Переменная `a` объявлена с явной декларацией типа. Декларация определяет тип переменной, тип выражения-инициализатора не должен ей противоречить, но не обязан совпадать. Переменная `b` объявлена без явного указания типа, поэтому ее тип определяется только типом выражения-инициализатора. Тип, содержащий переменный набор типовых аргументов, должен быть корректным вне зависимости от наличия или отсутствия явной декларации типа.

Анализ деклараций типа реализуется в классе `TypeResolver`. В процесс разрешения типа для класса добавляется дополнительный шаг, следующий после сбора объявленных пользователем типовых аргументов. Если последний типовой параметр класса, для которого создается тип, имеет переменную размерность, хвост набора типовых аргументов преобразуется. Собранные типы сохраняются в аннотацию @TypeArguments, а тип аргумента заменяется на тип верхней границы, если он есть в объявлении типового параметра, и на `Any?` в ином случае. Помимо этого проверяется, что тип корректно использует *-проекции, – такая проекция должна быть единственным аргументом, подставленным вместо набора параметров.

Теперь рассмотрим, как создается тип объекта при вызове конструктора. В качестве примера будем использовать класс кортежа

(см. рис. 3.10) и выражение создания переменной, приведенное на рис. 4.3. Для удобства здесь же повторим объявление конструктора класса.

```
1 class Tuple<Ts...> (vararg values: Ts) { /* ... */ }
2
3 val tuple: Tuple<Int, String> = Tuple<Int, String>(5, "value")
```

Рис.4.3. Создание кортежа

В примере используются все компоненты, участвующие в процессе вывода типа для вызова конструктора:

- В конструкторе указаны типовые аргументы;
- Переменный набор аргументов (`values`) в конструкторе имеет тип переменного набора типовых параметров (`Ts`);
- Декларация типа слева от знака равенства задает ожидаемый тип выражения.

Вывод типа конструктора является частью более общего процесса разрешения вызова функции. Задача этой процедуры – выбрать правильное объявление для вызова и с помощью системы уравнений вывести недостающие типы, а также проверить корректность зафиксированных типов. Вывод типов следует после выбора функции-кандидата, когда становится однозначно понятен список типовых параметров и аргументов. Изменения компилятора в данной работе не затрагивают процесс выбора кандидата, но влияют на вывод типов после его определения.

Несмотря на то, что внешне участок кода `Tuple<Int, String>` в примере выглядит абсолютно одинаково для декларации типа и для вызова конструктора, с точки зрения компиляции это объекты разных видов. Сопоставление параметров и аргументов для типов и для вызовов реализуется с помощью различных механизмов. При обработке деклараций типов разрешаются типовые проекции, состоящие из комбинации типа и вариантности, в то время как типовые аргументы вы-

зова представляют собой обычные типы. Соответственно, в обработке вызова конструктора класс `TypeResolver` не участвует. Вместо этого сопоставление аргументов хранится в объекте `TypeArgumentsMapping`, который можно воспринимать как ассоциативный массив, отображающий параметрый в аргументы. После изменения типа связи между параметрами и аргументами на один-ко-многим параметру сопоставляется список аргументов.

Класс `TypeArgumentsMapping` – не окончательное место фиксации типовых аргументов. Как было сказано в разделе 4.2, Kotlin поддерживает вывод типов, это относится и к не заданным явно типовым аргументам. Поэтому финальный тип типовых аргументов определяется системой ограничений, создаваемой для всего вызова конструктора, а информация из `TypeArgumentsMapping` служит для задания части уравнений системы.

На примере вызова конструктора `Tuple` (см. рис. 4.3) рассмотрим, какие изменения в системе уравнений позволяют обрабатывать переменный набор типовых параметров. Для начала приведем базовую систему, создаваемую компилятором без учета переменной размерности типового параметра.

$$\texttt{Ts} <: \texttt{Any?} \tag{4.1}$$

$$\texttt{Int} <: \texttt{Ts} \tag{4.2}$$

$$\texttt{String} <: \texttt{Ts} \tag{4.3}$$

$$\texttt{Tuple} < \texttt{Ts} > <: \texttt{Tuple} < \texttt{Any?} > \tag{4.4}$$

$$\texttt{Ts} <: \texttt{Int} \tag{4.5}$$

$$\texttt{Ts} <: \texttt{String} \tag{4.6}$$

Уравнение 4.1 определяется верхней границей типового параметра в объявлении конструктора. Поскольку явного ограничения нет, границей сверху является только тип `Any?`. Уравнения 4.2 и 4.3 –

это ограничения на тип однородного набора аргументов. Переменная `Ts` должна быть надтипом для всех аргументов однородного набора, т.к. аргументы объявлены с модификатором `vararg`. Уравнение 4.4 определяется ожидаемым типом переменной `tuple`. К моменту разрешения вызова ожидаемый тип типового параметра (`Any?`) создан классом `TypeResolver`, а типы `Int` и `String` сохранены в аннотации `@TypeArguements`. Два последних уравнения 4.5 и 4.6 задаются явно указанными в конструкторе типовыми аргументами. Можно заметить, что уравнения 4.2, 4.3 и 4.5 и 4.6 несовместны. Это не слишком удивительно с учетом того, что система уравнений пытается с помощью одной типовой переменной `Ts` выразить одновременно несколько разнородных типов.

Чтобы устранить противоречие, для каждого типового аргумента заводится отдельная типовая переменная, а общая переменная `Ts` используется как наиболее конкретный общий тип для всех типовых аргументов. Для этого знак в уравнениях 4.5 и 4.6 меняется на противоположный, чтобы отношение переменной `Ts` и явных типовых аргументов повторяло отношение с типами обычных аргументов функции. В итоге имеем набор дополнительных уравнений 4.7 – 4.14 .

$$\texttt{Int} <: \texttt{Ts} \tag{4.7}$$

$$\texttt{String} <: \texttt{Ts} \tag{4.8}$$

$$\texttt{Int} <: \texttt{Ts}[0] \tag{4.9}$$

$$\texttt{Int} = \texttt{Ts}[0] \tag{4.10}$$

$$\texttt{String} <: \texttt{Ts}[1] \tag{4.11}$$

$$\texttt{String} = \texttt{Ts}[1] \tag{4.12}$$

$$\texttt{Ts}[0] = \texttt{Int} \tag{4.13}$$

$$\texttt{Ts}[1] = \texttt{String} \tag{4.14}$$

Уравнения 4.7 и 4.8 – исправленные версии уравнений 4.5 и 4.6.

Уравнения 4.9 и 4.11 выражают ограничения, накладываемые разнородным набором аргументов, 4.10 и 4.12 – явными типовыми аргументами, а 4.13 и 4.14 – ожидаемым типом. Знаки равенства в дополнительных уравнениях определяются поддержкой в новом синтаксисе только инвариантных проекций типа. Решение системы 4.1 – 4.4, 4.7 – 4.14 дает необходимые результаты:

`Ts` = `Any`?

`Ts`[`0`] = `Int`

`Ts`[`1`] = `String`

Последний шаг создания типа для вызова конструктора состоит в записи выведенных типов переменных из системы уравнений в тип вызова. Для переменной `Ts` дополнительных действий не требуется, поскольку она сопоставлена существующему типовому параметру в объявлении и обрабатывается без искусственного вмешательства. Дополнительные проиндексированные типовые переменные `Ts[0]` и `Ts[1]` не привязаны к типовым параметрам, поэтому требуют специальной обработки. Опуская детали реализации можно сказать, что в момент формирования окончательного представления для вызова выведенные типы помещаются в синтетичесую аннотацию `@TypeArguments`, которая затем дополняет тип вызова.

4.4.3. Работа с индексированными типами

Мы рассмотрели процесс сохранения типовых аргументов в аннотированный тип, а также проверку разнородного набора аргументов. Теперь покажем, как с использованием сохраненной информации происходит обработка индексированных типов.

Типовой аргумент функции становится индексированным типом после добавления дополнительного ограничения в систему уравнений для вызова. Новое уравнение фиксирует равенство типового аргумента функции и i-го типового аргумента объекта, на котором она вызвана, где i – это индекс, аргумент функции. Объявление аннота-

ции `@TypeIndex`, делающей аругумент функции индексом, показано на рис. 4.4. Пример ее использования был показан на рис. 3.10.

```
1 @Target(AnnotationTarget.TYPE)
2 @Retention(AnnotationRetention.SOURCE)
3 annotation class TypeIndex(
4     val variadicParameter: kotlin.reflect.KClass<*>,
5     val targetType: kotlin.reflect.KClass<*>
6 )
```

Рис.4.4. Аннотация @TypeIndex

Формирование дополнительного уравнения происходит во время разрешения вызова функции и включает следующие действия:

- Среди аргументов функции по наличию аннотации `@TypeIndex` определяются индексы;
- По содержимому аннотации вычисляется, какой типовой параметр функции станет типом, проиндексированным данным аргументом;
- В типе объекта, на котором вызвана функция, производится поиск аннотации @TypeArguments, хранящей значения типовых аргументов;
- По индексу и списку типовых аргументов объекта вычисляется индексированный тип;
- В систему уравнений добавляется ограничение на равенство типовой переменной функции и значения индексированного типа, полученного на предыдущем шаге.

Данных операций достаточно для вывода индексированного типа при корректном объявлении и использовании функции. Поиск ошибок происходит отдельно в дополнительной проверке вызова `VariadicMethodCallChecker`. Она реализует интерфейс проверок вызовов `CallChecker` аналогично тому, как для это делается для объявлений с помощью интерфейса `DeclarationChecker`. В проверке анализируется, что функция вызывается с индексом корректного типа,

объект, на котором совершается вызов, содержит аннотацию со списком типовых аргументов, а значение индекса не выходит за границы списка.

Существует случай, когда испльзование переменной в качестве значения индекса не является ошибкой. Этот случай – вызов функции с индексированным типом из другой подобной функции с передачей индекса по цепочке. В таком варианте значение индексированного типа не вычисляется обычным способом. Вместо этого добавляется дополнительное уравнение, связывающее между собой типовые переменные двух индексированных типов. Благодаря этому индексированный тип, выведенный в момент окончательной подстановки индекса, повлияет на все индексированные типы в цепочке вызовов, и информация о типах не будет потеряна.

Таким образом, обработка типовых параметров переменного размера проходит полностью во фронтенде компилятора. Статическая типизация переменного набора типовых аргументов заканчивается на этапе анализа.

5. ТЕСТИРОВАНИЕ И ОЦЕНКА РЕЗУЛЬТАТОВ

5.1. Тестирование

При тестировании компилятора внимание уделялось двум основным вопросам:

- Корректности работы нового синтаксиса;
- Работоспособности старого синтаксиса.

Для проверки были использованы диагностические тесты компилятора и тесты кодогенерации. Тесты кодогенерации выполняют запуск проверяемой программы на целевом бекенде и работают по принципу «черного ящика». Диагностические тесты проверяют способность компилятора находить ошибки в некорректном коде и отсут-

ствие ложных диагностик для корректного кода. Один диагностический тест – это исходный код программы на Kotlin с дополнительной разметкой ожидаемых ошибок, которые должен обнаружить компилятор. Пример теста показан на рис. 5.1.

```
1  // !LANGUAGE: +NewInference
2  // !DIAGNOSTICS: -UNUSED_VARIABLE -UNCHECKED_CAST
3
4  import kotlin.experimental.*
5
6  class Tuple<Ts...> (vararg values: Ts) {
7      private val _values: Array<Any?> = arrayOf(*values)
8      fun <T> get(ix: @TypeIndex(Ts::class, T::class) Int): T =
           _values[ix] as T
9  }
10
11 fun test() {
12     val tuple = Tuple<Int, String>(42, "42")
13     val first: Int = tuple.get(0)
14     val second: Double =
15         <!TYPE_MISMATCH!>tuple.<!TYPE_MISMATCH!>get(1)<!><!>
16 }
```

Рис.5.1. Диагностический тест

Разметка <!>*тип_ошибки*<!>...<!> отмечает позицию ожидаемых диагностик. В примере делается попытка сохранить вторую переменную кортежа с типом `String` в переменную с типом `Double`, это ошибка вида `TYPE_MISMATCH`.

Всего написано 26 дополнительных диагностических тестов для проверки разных случаев использования нового синтаксиса. При запуске старых диагностических тестов ошибок, вызванных нарушением обратной совместимости, найдено не было. Новые тесты содержат следующие классы проверок[1]:

- Вывод типов при комбинировании явного и неявного задания типовых аргументов;
- Обработка вложенных типовых аргументов;

[1] Полный список тестов доступен в репозитории `https://github.com/KirpichenkovPavel/kotlin/tree/master/compiler/testData/diagnostics/testsWithStdLib/variadicGenerics`

- Проверка вывода типов для функций с индексированными типами;
- Сочетание индексированных типов с операторными функциями;
- Поведение *-проекций и явных преобразований типов при использовании переменного набора типовых параметров.

Тестированию кодогенерации было уделено меньше внимания, т.к. дополнительная информация о типах используется только во время анализа и не сохраняется в исполняемом коде. Тестирование заключалось в запуске существующих тестов, все тесты успешно проходят на модифицированной версии компилятора.

5.2. Оценка результатов и дальнейшее развитие

Предложенный в данной работе вариант синтаксиса покрывает сценарии использования для классов с переменным набором типовых параметров и понятной логикой обработки одного элемента. Кортеж и вариант являются примерами таких классов. Пример их возможного определения и использования привиден в Приложении.

Применение аннотаций для обозначения индекса – компромисс, сделанный в прототипной версии компилятора. Полноценная реализация могла бы включать дополнительные конструкции, такие как оператор индексирования над типовым параметром переменного размера. Однако подобная реализация была бы значительно более трудоемкой, поэтому в рамках данной работы от нее было решено отказаться.

В работе также допущен ряд упрощений, ограничивающий функциональные возможности прототипа компилятора:

- Поддерживается только инвариантный типовой параметр переменной размерности. Поддержка ко- и контрвариантности затронет систему уравнений при использовании индексированных типов и потому требует дополнительного анализа;

- Не рассматривается случай наследования классов с типовыми параметрами переменного размера;
- Как следствие запрета на наследование, не рассматривается случай переопределения функции с переменным набором аргументов.

Устранение этих упрощений сделает возможной единую реализацию интерфейса для типов функций в Kotlin, о котором было сказано в разделе 1.2. Пример возможной реализации единого интерфейса `Function` показан на рис. 5.2.

```
1  interface Function<out R, in Ts...> {
2      operator fun invoke(vararg args: Ts): R
3  }
4
5  val MyFunction = object: Function<ReturnType, Type1, Type2> {
6      override operator fun invoke(
7          arg1: Type1,
8          arg2: Type2
9      ): ReturnType {
10         // ...
11     }
12 }
```

Рис.5.2. Интерфейс Function

В текущем состоянии вариантность типовых параметров не будет обработана корректно. Функция invoke, объявленная для фиксированного количества аргументов при реализации интерфейса `Function`, не будет воспринята как корректное переопределение из-за отличия в аргументах. Названные проблемы являются ограничениями прототипа и могут быть устранены в дальнейшем.

Помимо этого существуют сценарии использования переменного набора типовых параметров, не имеющие очевидных способов выражения в предложенном синтаксисе. Это вызвано тем, что типы параметров не могут быть преобразованы совместно, даже если логика такого преобразования носит обобщенный характер. Сложность представляет запись типа, получаемого в результате трансформации. Например, операция конкатенации кортежей имеет понятную логику –

объединение двух наборов элементов в один. Но для выражения типа итогового кортежа требуется операция объединения наборов типовых параметров. Поскольку в текущей версии Kotlin операции над типами не поддерживаются, а потенциальный набор преобразований неограничен, выбор допустимых преобразований для типовых переменных представляется непростой задачей.

Дальнейшее развитие данной работы включает два возможных направления: устранение ограничений прототипа и добавление операций над набором типовых параметров. При успешном решении этих задач новый синтаксис позволит полноценно выражать переменный набор типовых параметров.

ЗАКЛЮЧЕНИЕ

В работе были проанализированы существующие подходы к реализации параметрического полиморфизма с переменным набором типовых параметров в различных языках программирования. В результате анализа сделан вывод, что использование данных подходов в Kotlin требует их существенной переработки, поскольку в языке не используются зависимые типы и шаблонная генерация кода для полиморфных объектов. В качестве примеров использования переменного набора типовых параметров рассмотрены кортежи, варианты и функции. Предложен дизайн синтаксиса, поддерживающего переменный набор типовых параметров посредством индексированных типов. Поддержка измененного синтаксиса реализована в прототипной версии компилятора. По результатам тестирования прототипа сделаны выводы о возможностях и ограничениях предложенного синтаксиса. Главное выявленное ограничение нового синтаксиса состоит в отсутствии механизма для преобразования переменного набора типовых параметров, потребность в котором возникает на практике.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Cardelli Luca, Wegner Peter. On understanding types, data abstraction, and polymorphism // ACM Computing Surveys (CSUR). — 1985. — Vol. 17, no. 4. — P. 471–523.
2. Gregor Douglas P, Järvi Jaakko. Variadic Templates for C++ 0x. // Journal of Object Technology. — 2008. — Vol. 7, no. 2. — P. 31–51.
3. JFlex home page [Электронный ресурс]. — URL: `https://jflex.de/` (дата обращения: 22.05.2019).
4. The Java Language Specification, Java SE 8 Edition / James Gosling, Bill Joy, Guy L. Steele et al. — 1st edition. — Addison-Wesley Professional, 2014. — ISBN: 013390069X, 9780133900699.
5. Magalhães José Pedro, Löh Andres. A formal comparison of approaches to datatype-generic programming // arXiv preprint arXiv:1202.2920. — 2012.
6. Program Structure Interface (PSI) [Электронный ресурс], IntelliJ Platform SDK DevGuide. — URL: `https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html` (дата обращения: 04.06.2019).
7. Reference for std::variant [Электронный ресурс], C++ reference. — URL: `https://en.cppreference.com/w/cpp/utility/variant` (дата обращения: 04.06.2019).
8. Variable-Arity Polymorphism : Rep. / Technical Report NU-CCIS-08-03, Northeastern University ; Executor: T Stephen Strickland, Sam Tobin-Hochstadt, Matthias Felleisen : 2008.
9. Stroustrup Bjarne. The C++ programming language. — Pearson Education India, 2000.
10. Templates in D [Электронный ресурс], D language specification. — URL: `https://dlang.org/spec/template.html` (дата обращения: 07.05.2019).
11. TypeScript definitions for library Reactive Extensions [Элек-

тронный ресурс], GitHub. — URL: https://github.com/Reactive-Extensions/RxJS/blob/master/ts/rx.async.d.ts (дата обращения: 07.05.2019).

12. Venners Bill. The java virtual machine. — McGraw-Hill, New York, 1998.
13. Wadler Philip, Blott Stephen. How to make ad-hoc polymorphism less ad hoc // Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages / ACM. — 1989. — P. 60–76.
14. Weirich Stephanie, Casinghino Chris. Arity-generic datatype-generic programming // Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification / ACM. — 2010. — P. 15–26.
15. Xi Hongwei. Imperative programming with dependent types // Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332) / IEEE. — 2000. — P. 375–387.

Приложение

ЛИСТИНГИ

Листинг 1. Класс Tuple и пример его использования

```
class Tuple<Ts...> (vararg values: Ts) {
    private val _values: Array<Any?> = arrayOf(*values)

    operator fun <T> get(
        ix: @TypeIndex(Ts::class, T::class) Int
    ): T = _values[ix] as T

    fun <T, R> let(
        ix: @TypeIndex(Ts::class, T::class) Int,
        block: Tuple<Ts>.(value: T) -> R
    ): R = this.block(_values[ix] as T)

    fun asAnyList(): List<Any?> = _values.toList()
}

fun <T, Ts...: T> Tuple<Ts>.asList(): List<T> =
    asAnyList().map { it as T }

fun example() {
    val myTuple = Tuple(15, 86.0, "value")
    val myDouble: Double = myTuple[1]
    tuple.let(2) { string -> println(string.substring(3)) }

    val listOfNumbers: List<Number> = Tuple(3, 14.6).asList()
}
```

Листинг 2. Класс Variant и пример его использования

```
class Variant<Ts...> () {
    private var value: Any? = null
    private var index: Int = -1

    operator fun <T> set (
        ix: @TypeIndex(Ts::class, T::class) Int,
        value: T
    ) {
        this.value = value
        index = ix
    }

    fun <T> with (
        ix: @TypeIndex(Ts::class, T::class) Int,
        value: T
```

```
    ): Variant<Ts> {
        set(ix, value)
        return this
    }

    operator fun <T> get (
        ix: @TypeIndex(Ts::class, T::class) Int
    ): T? {
        if (index == -1) {
            // not initialized - throw exception
            throw java.lang.Exception("not initialized")
        }
        if (ix != index) {
            return null
        }
        return value as T
    }

    fun <T, R> let (
        ix: @TypeIndex(Ts::class, T::class) Int,
        block: (T) -> R
    ): R? = get(ix)?.let(block)
}

data class Leaf(val data: String)
data class Node(
    val left: Variant<Node, Leaf>,
    val right: Variant<Node, Leaf>,
    val data: String
)

fun collectValues(tree: Variant<Node, Leaf>): String {
    var result = ""
    tree.let(0) { node ->
        result = """
            ${collectValues(node.left)}
            ${node.data}
            ${collectValues(node.right)}
        """.trimIndent()
    }
    tree.let(1) { leaf ->
        result = leaf.data
    }
    return result
}

```

```
62 fun hello() {
63     val leftLeaf = Variant<Node, Leaf>().with(1, Leaf("Hello"))
64     val rightLeaf = Variant<Node, Leaf>().with(1, Leaf("World!"))
65     val root = Variant<Node, Leaf>().with(
66         0, Node(leftLeaf, rightLeaf, ", ")
67     )
68     println(collectValues(root))
69 }
```