Санкт-Петербургский государственный университет

Поляков Александр Романович

Выпускная квалификационная работа

Декларативный UI DSL для разработки мобильных приложений

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2019 «Программная инженерия»*

Научный руководитель:
к.ф.-м.н., доцент Д.В. Луцив

Консультант:
к.ф.-м.н., руководитель направления разработки языков программирования
ООО "Техкомпания Хуавей" А.Е. Недоря

Рецензент:
инженер-программист ООО "Техкомпания Хуавей" Д.И. Соломенников

Санкт-Петербург
2021

Saint Petersburg State University

Alexander Romanovich Polyakov

Master's Thesis

Declarative UI DSL for mobile applications development

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2019 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D.V. Luciv

Consultant:
C.Sc., head of programming languages department at Huawei Technologies Co. Ltd.
A.E. Nedorya

Reviewer:
software engineer at Huawei Technologies Co. Ltd. D.I. Solomennikov

Saint Petersburg
2021

Оглавление

Введение

Жизнь современного человека невозможно представить без мобильных устройств, проникших практически во все сферы его жизни. Речь идет о смартфонах, планшетах, "умных часах" и так далее. При этом человеко-машинное взаимодействие осуществляется через специальный вид программного обеспечения — мобильные приложения. Графический интерфейс является важной компонентой мобильных приложений, поскольку непосредственно влияет на их доступность и удобство использования.

Рынок мобильных устройств и приложений в последние годы непрерывно растет [17,19], также возрастает и разнообразие архитектур процессоров и операционных систем мобильных устройств [2,10]. Это разнообразие, а также необходимость оптимизировать процесс разработки мобильных приложений стали причиной интенсивного развития средств создания мобильных приложений [5,8,12,20,25,26].

На сегодняшний день средства разработки мобильных приложений представляют собой многокомпонентные программные и/или программно-аппаратные комплексы, включающие среду разработки на некотором языке программирования, компилятор, отладчик, среду времени выполнения для программ, написанных на этом языке, подсистему отрисовки графического интерфейса, отладочные платы целевых устройств и их эмуляторы и так далее.

Несмотря на кажущуюся «обыденность», создание и отрисовка графического интерфейса приложения являются одними из наиболее сложных задач, с которыми сталкиваются разработчики мобильных приложений. Использование декларативных языков для описания интерфейсов и увеличение роли компилятора в процессе отображения пользовательского интерфейса на экране являются тенденциями последних лет в данной области.

Accord — язык программирования общего назначения, сочетающий элементы компонентного, объектно-ориентированного и функционального программирования. Данный язык разрабатывается в Санкт-Петербургском научно-исследовательском центре компании Huawei. На

текущий момент язык находится в стадии ранней разработки и не обладает устоявшейся спецификацией. Прототип компилятора языка написан на языке программирования Go [6], имеет автоматически генерируемый по формальной контекстно-свободной грамматике в расширенной форме Бэкуса-Наура [15] и ручной LL [21] парсеры, оптимизатор, работающий над высокоуровневым промежуточным представлением программы, кодогенерацию в LLVM IR [14] и байткод некоторой виртуальной машины. В будущем спецификация языка Accord и исходный код его компилятора станут открытыми.

Одно из перспективных направлений применения языка Accord — разработка мобильных приложений. На данный момент, универсальный характер языка и реализации его компилятора не позволяют оптимально реализовать необходимую для мобильной разработки функциональность, связанную с программированием интерфейса пользователя. Данная работа фокусируется на внесении изменений в спецификацию и компилятор языка Accord, позволяющих использовать его в качестве языка разработки мобильных приложений, не уступающего существующим аналогичным решениям.

1. Постановка задачи

Целью данной работы является внесение изменений в спецификацию языка Accord и его компилятор, позволяющих декларативно описывать пользовательский интерфейс приложений, а также использовать статическую информацию о программе в процессе отрисовки графического интерфейса. Для достижения этой цели были поставлены следующие задачи:

- выполнить обзор предметной области и существующих решений;
- выполнить сбор и анализ требований к современному языку разработки мобильных приложений и его компилятору;
- предложить изменения, которые необходимо внести в спецификацию языка программирования Accord и его компилятор для достижения поставленной цели;
- реализовать данные изменения;
- провести апробацию полученного решения.

2. Обзор

В данном разделе представлен обзор предметной области, включающий: общие сведения; способы реализации предметно-ориентированных языков вместе с преимуществами и недостатками каждого из подходов; процесс отображения пользовательского интерфейсов в упрощённом виде; популярные языки разработки мобильных приложений.

2.1. Общие сведения

2.1.1. Завершающие лямбда-выражения

Завершающие лямбда-выражения (от англ. *trailing lambdas*) — синтаксические возможности некоторых языков программирования, позволяющие при вызове функции, последним формальным параметром которой является другая функция, передать в качестве последнего аргумента лямбда-выражение, вынесенное за пределы синтаксического контекста остальных аргументов. Добавление в язык завершающих лямбда-выражений позволяет на уровне синтаксиса сделать неотличимыми конструкции языка программирования и вызов вышеупомянутых функций.

На листинге 1 представлен пример синтаксического правила передачи завершающих лямбда-выражений, записанный в нотации инструмента ANTLR4 [1].

```
trailing_lambda
  : '{' parameters? (return_type 'in')? expression_sequence '}'
  ;

function_call
  : function_name arguments? trailing_lambda?
  ;
```

Листинг 1: Синтаксическое правило завершающих лямбда-выражений

2.1.2. Непрозрачные типы

Непрозрачные типы (от англ. *opaque types*) — механизм типовой системы языка программирования и его компилятора, позволяющий функции или методу с непрозрачным типом возвращаемого значения скрывать информацию о типе возвращаемого значения. Вместо того, чтобы указывать конкретный тип в качестве типа возвращаемого значения функции, тип возвращаемого значения описывается в терминах интерфейсов, которым он соответствует. В отличие от возврата значения, тип которого является типом интерфейса, непрозрачные типы сохраняют идентичность — компилятор имеет доступ к информации о конкретном типе, в то время как клиенты функции или метода его не имеют.

2.2. Предметная область

2.2.1. Предметно-ориентированные языки

Предметно-ориентированный язык (*domain-specific language*, DSL) — это язык программирования с более высоким уровнем абстракции, отражающий специфику решаемых с его помощью задач. Такой язык оперирует понятиями и правилами из определенной предметной области [9].

В отличие от языков программирования общего назначения, таких как C, Python, Java, предметно-ориентированные языки предоставляют абстракции, адекватные решаемой проблеме, позволяя выражать решения, написанные с их помощью, кратко и ёмко; причём в некоторых случаях использование DSL не требует квалификации программиста. В качестве примера DSL можно привести SQL — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных. Основным недостатком применения предметно-ориентированных языков является стоимость их разработки, требующая экспертизы как в области разработки языков программирования, так и в целевой предметной области. Это является одной из причин того, что предметные языки редко применя-

ются для решения задач программной инженерии, в отличие от языков программирования общего назначения. Другой причиной отказа от обособленных предметных языков является тот факт, что сочетание программной библиотеки и языка программирования общего назначения может заменять DSL. Программный интерфейс (*Application Programming Interface*, API) библиотеки содержит специфичный для определённой области словарь, образованный именами классов, методов и функций, доступный всем пользователям языков программирования общего назначения, подключившим библиотеку. Однако, вышеприведённый подход проигрывает предметным языкам в следующих аспектах [16, 27]:

- устоявшаяся в области нотация, как правило, выходит за рамки ограниченных механизмов определения пользовательских операторов, предоставляемых языками общего назначения;
- абстракции определённой области не всегда могут быть просто отображены в конструкции языков общего назначения [7];
- использование предметно-ориентированного языка сохраняет возможность анализа, верификации, оптимизации, параллелизации и трансформации в рамках конкретной области, что, в случае работы с исходным текстом языка программирования общего назначения, является более сложной задачей.

2.2.2. Подходы к реализации предметно-ориентированных языков

В последнее время всё больше исследований в области предметно-ориентированных языков направлены на категоризацию предметных языков, а также выработку советов и лучших практик, отвечающих на вопросы "когда и как?" создавать DSL для конкретной области [16, 22, 28].

Препроцессинг DSL-конструкции транслируются в более низкоуровневый программный код базового языка программирования общего на-

значения.

- *Макрокоманда.* Конструкции предметного языка представлены символьными именами, заменяемыми при обработке препроцессором на последовательность программных инструкций базового языка.

- *Транспиляция.* Исходный код предметного языка транслируется в исходный код языка общего назначения.

- *Лексическая обработка.* Трансформация предметного языка в язык общего назначения осуществляется на уровне лексем.

Преимуществом данного подхода является простота реализации DSL, поскольку большая часть семантического анализа выполняется средствами базового языка. В то же время, это является и недостатком данного подхода ввиду отсутствия предметно-ориентированного статического анализа, оптимизаций и сообщений об ошибках.

Встраивание в базовый язык В данном подходе конструкции базового языка используются для построения библиотеки предметно-ориентированных операций. С помощью синтаксиса базового языка задаётся диалект, максимально приближенный к определённой предметной области.

Преимуществом данного подхода является полное переиспользование компилятора или интерпретатора базового языка для построения DSL. Основными недостатками являются сообщения об ошибках, соответствующие спецификации базового языка, и ограниченная синтаксическая выразительность, обусловленная существующим синтаксисом базового языка.

Самостоятельный компилятор В данном подходе для создания DSL используются методы построения компиляторов или интерпретаторов. В случае компилятора, конструкции предметного языка транслируются во внутреннее представление компилятора, а статический

анализ производится над спецификацией DSL. В случае интерпретатора, конструкции предметного языка распознаются и выполняются в ходе цикла выборки-распознавания-исполнения (от англ. *fetch-decode-execute cycle*).

Преимуществами данного подхода являются приближенные к предметной области синтаксис языка и сообщения об ошибках. Серьёзным недостатком является необходимость создания нового компилятора или интерпретатора предметного языка.

Компилятор компиляторов Данный подход схож с предыдущим за исключением того, что все или некоторые стадии компиляции выполняются с использованием *компилятора компиляторов* — программы, воспринимающей синтаксическое или семантическое описание языка программирования и генерирующей компилятор для этого языка.

Преимуществом подхода является снижение расходов на создание компилятора предметного языка. Ограниченность итогового DSL возможностями используемого компилятора компиляторов, а также сложность проработки предметного языка в деталях, что может быть критично для достижения определённого уровня производительности и близости сообщений об ошибках к предметной области, составляют недостатки данного подхода.

Расширение существующего компилятора Компилятор языка программирования общего назначения расширяется предметно-ориентированными правилами оптимизации и/или генерации кода.

В сравнении с предыдущим, данный подход менее трудоёмок из-за возможности переиспользования частей существующего компилятора. Однако, стоит отметить, что расширение существующего компилятора может оказаться сложной задачей, для выполнения которой необходима поддержка расширений со стороны компилятора языка общего назначения, а также минимизация пересечений синтаксиса и семантики базового и предметного языков.

Использование готовых инструментов Существующие инструменты и нотации адаптируются под конкретную предметную область. Примером такого подхода являются DSL, основанные на нотации XML. В большинстве случаев предметные языки, полученные данным способом, плохо подходят для их использования людьми в ручном режиме.

Создание библиотеки функций В качестве предметного языка выступает язык программирования общего назначения и библиотека функций, создающая своим интерфейсом словарь предметной области.

Преимуществом подхода является его относительная простота, заключающаяся в необходимости создания библиотеки функций предметной области. Недостатком является сложность работы с таким языком для людей, не обладающими навыками программирования.

2.2.3. Отображение пользовательского интерфейса

В данном разделе описан процесс обновления пользовательского интерфейса, ставший де-факто стандартом для популярных современных средств разработки мобильных приложений. Стоит отметить, что далее будут рассмотрены лишь базовые принципы построения и отображения интерфейсов, которых должно быть достаточно для объяснения тех или иных оптимизационных решений, принятых в данной работе.

Задачей отображения пользовательского интерфейса занимается так называемый движок рендеринга или отрисовки (*rendering engine*) — программное обеспечение, генерирующее изображение по модели. Здесь *модель* — это описание любых объектов или явлений на строго определённом языке или в виде структуры данных.

Процесс отображения пользовательского интерфейса (рис. 1) состоит из последовательного построения нескольких древовидных структур, каждая из которых является образом графического интерфейса, записанного пользователем на некотором языке.

- *Дерево компонентов* — древовидная структура данных, узлами которой являются высокоуровневые представления графических

компонент. Множество операций над деревом компонентов анализируется и оптимизируется, после чего применяется к дереву элементов, изменение которого является дорогостоящей операцией. Примером такого дерева может являться VDOM (от англ. *Virtual Document Object Model* — виртуальная объектная модель документа).

- *Дерево элементов* — древовидная структура данных, изоморфная дереву компонентов, элементы которой трансформируются в элементы дерева рендеринга для последующей их отрисовки графическим конвейером. В качестве примера такого дерева можно привести DOM (от англ. Document Object Model).

- *Дерево рендеринга* — древовидная структура данных, содержащая в себе только необходимую для отрисовки информацию. Элементами этого дерева являются базовые графические примитивы, доступные движку отрисовки.

При отрисовке следующего кадра, движок рендеринга перерисовывает только изменившиеся по сравнению с предыдущим кадром части интерфейса. Для этого, между двумя соседними кадрами находится разница, которая и отправляется в движок отрисовки. Нахождение этой разницы является одной из наиболее ресурсоёмких операций во время выполнения приложения, поскольку она требует обхода дерева элементов интерфейса, нахождения узлов, соответствующих потенциально изменяющимся компонентам графического интерфейса, сравнения таких компонент текущего и предыдущего кадров.

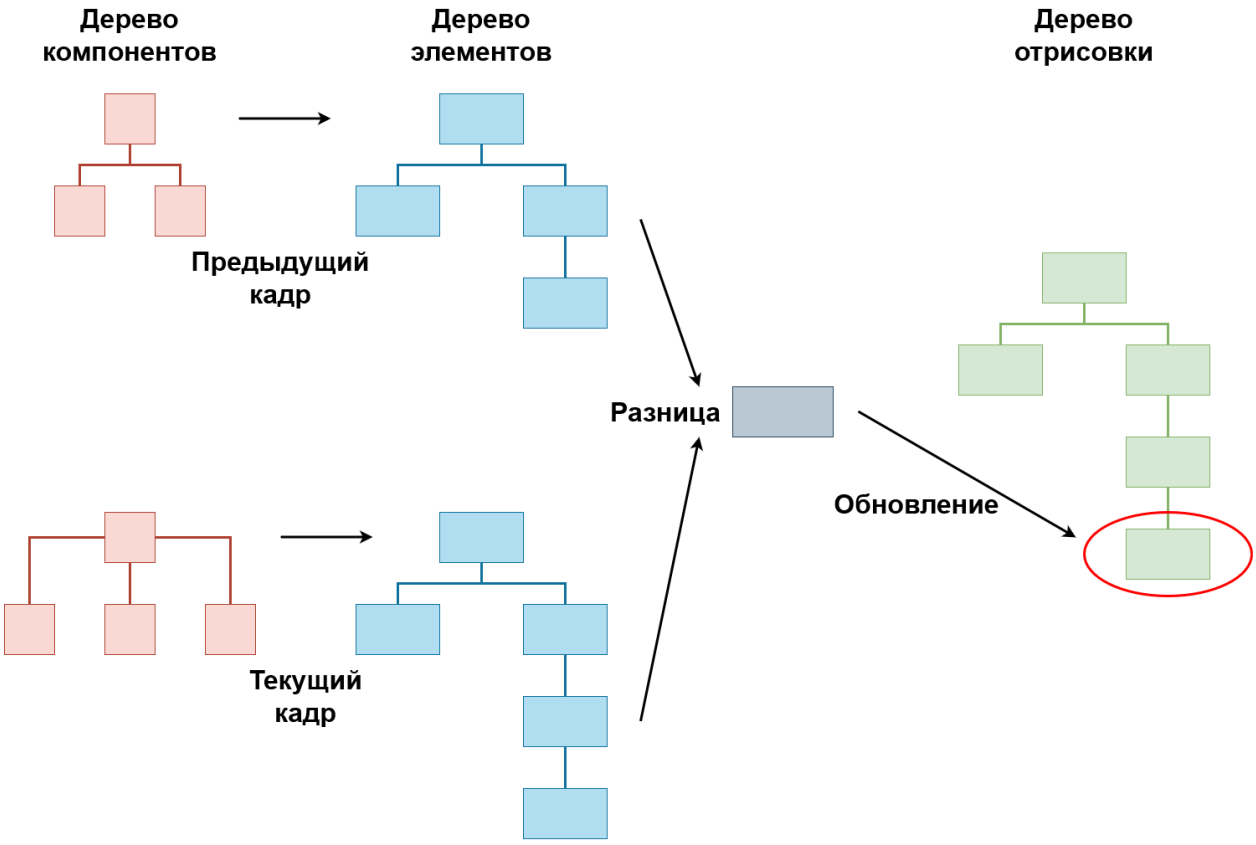

Рис. 1: Процесс отображения пользовательского интерфейса

2.3. Существующие решения

Dart / Flutter

Flutter [5] — язык разработки мобильных приложений, представляющий из себя создающую словарь предметной области библиотеку графических компонентов для языка программирования общего назначения Dart [4].

На листинге 2 представлен пример исходного кода приложения, написанного на языке Flutter. Структура графического интерфейса описывается декларативно (строки 17-25). Управление реактивными данными вынесено в отдельный класс и осуществляется пользователем в явном виде: оборачивание всех реактивных изменений в вызов специальной функции *setState*, в качестве аргумента которой передаётся лямбда-выражение, содержащее часть бизнес-логики. При вызове этой функции, выполняется переданное ей лямбда-выражение, после чего

графическая компонента помечается как компонента, требующая дальнейшего обновления.

```
class Counter extends StatefulWidget {
    Counter({Key key}) : super(key: key);

    @override
    _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    int _counter = 0;

    void _incrementCounter() {
        setState( () { _counter++; } );
    }

    @override
    Widget build(BuildContext context) {
        return Column(
            children: <Widget>[
                Text(''Current count: $_counter''),
                TextButton(
                    onPressed: _incrementCounter,
                    child: Text(''Click on me!'')
                )
            ]
        )
    }
}
```

Листинг 2: Flutter. Счётчик нажатия кнопки

Отрисовка графического интерфейса средством разработки мобильных приложений Flutter осуществляется по описанному в пункте 2.2.3 алгоритму.

Kotlin DSL / Jetpack Compose

Kotlin DSL — механизм языка Kotlin [11], позволяющий создавать на его основе предметно-ориентированные языки. Завершающие лямбда-выражения 2.1.1 и тот факт, что результатом последовательности выражений является её последнее выражение, суть основные техники, на основе которых реализован Kotlin DSL.

На листинге 3 представлен пример исходного кода приложения, написанного с использованием фреймворка для разработки мобильных приложений Jetpack Compose [8], реализованного в виде надстройки над Kotlin DSL. Структура графического интерфейса описывается декларативно (строки 5-10). Управление реактивными данными скрыто от пользователя и не является частью языка и компилятора. Работа с реактивными данными реализована в виде библиотеки функций, например, *remember* и *mutableStateOf* (строка 3).

```
1  @Composable
2  fun Counter() {
3      var count by remember { mutableStateOf(0) }
4
5      Column() {
6          Text(text = "Current count: ${count.value}")
7          Button(onClick = { count++ }) {
8              Text(text = "Click on me!")
9          }
10     }
11 }
```

Листинг 3: Jetpack Compose. Счётчик нажатия кнопки

Отрисовка графического интерфейса любым средством разработки мобильных приложений, построенным на базе Kotlin DSL, осуществляется по описанному в пункте 2.2.3 алгоритму.

JavaScript / React Native

React Native [20] — фреймворк для разработки мобильных приложений на языке программирования общего назначения JavaScript. С точки зрения классификации предметно-ориентированных языков, React Native является языком разработки мобильных приложений, реализованным в виде библиотеки графических компонент.

На листинге 4 представлен пример исходного кода приложения, написанного с использованием фреймоворка React Native. Структура графического интерфейса описывается в декларативном стиле (строки 8-16). Управление реактивными данным производится пользователем в явном виде: все операции над реактивными данными оборачиваются в

вызов предопределённой функции *setState*, которая по своему поведению аналогична такой же функции в языке Flutter.

```
class Counter extends React.Component {
  state = { count: 0 };

  incrementCounter = () => this.setState(
    prevState => ({ ...prevState, count: this.state.count + 1 })
  )

  render() {
    const { count } = this.state;
    return (
      <View>
        <Text>Current count: {count}</Text>
        <Button title="Click on me!" onPress={this.incrementCounter}/>
      </View>
    );
  }
}
```

Листинг 4: React Native. Счётчик нажатия кнопки

Отрисовка графического интерфейса осуществляется по алгоритму, описанному в пункте 2.2.3.

Swift / SwiftUI

SwiftUI [25] — язык разработки мобильных приложений, реализованный методом встраивания в базовый язык, которым, в данном случае, выступает язык программирования общего назначения Swift [23].

На листинге 5 представлен пример исходного кода приложения, написанного на языке SwiftUI. Структура графического интерфейса описывается декларативно (строки 4-11). Реактивные данные помечаются пользователем декоратором *@State*. Компилятор автоматически генерирует код, приводящий к обновлению графической компоненты в случае изменения реактивных данных.

```
struct Counter : View {
    @State var count: Int = 0

    var body: some View {
        VStack {
```

```
            Text("Current count: \(count)")
            Button(action: { self.count += 1 }) {
                Text("Click on me!")
            }
        }
    }
}
```

Листинг 5: Счётчик нажатия кнопки на языке Swift/SwiftUI

С точки зрения синтаксиса, для достижения декларативности описания интерфейса, в язык программирования Swift были добавлены следующие возможности и языковые конструкции:

- завершающие лямбда-выражения 2.1.1;
- неявный возврат значений из функций, состоящих из единственного выражения возврата результата;
- непрозрачные типы (ключевое слово *some*) 2.1.2

Особенностью SwiftUI является высокая степень использования информации, доступной во время компиляции программы, для оптимизации процесса отображения пользовательского интерфейса. Каждая графическая компонента обладает статическим типом. Так, компонента *Counter* имеет следующий тип:

```
struct {
    count: Int
    body: VStack[Text, Button[Text]]
}
```

Зная тип каждой графической компоненты, процесс отображения интерфейса может быть оптимизирован. Так, вместо алгоритма, описанного в пункте 2.2.3, SwiftUI делает лишь точечные вызовы обновления только тех компонент, которые могут потенциально измениться во время работы приложения. Преимуществом такого подхода является уменьшение потребления ресурсов процессора мобильным приложением во время обновления кадра. При обновлении кадра пропадает необходимость в обходе дерева компонент и проверки каждой компоненты на изменяемость. Все оптимизированные процедуры обновления

графических компонент генерируются компилятором автоматически во время компиляции приложения. Как именно SwiftUI использует статическую информацию о компонентах для выполнения вышеописанной оптимизации неизвестно ввиду проприетарности и закрытости данного решения.

3. Требования к спецификации и компилятору языка разработки мобильных приложений

В данном разделе представлены требования к современному средству разработки мобильных приложений, собранные на основе обзора существующих решений, а также сводная таблица соответствия существующих решений собранным требованиям. Предъявляемые требования были разделены на две группы: функциональные и нефункциональные.

3.1. Функциональные требования

Оптимизация отрисовки графического интерфейса

Отрисовка графического интерфейса мобильного приложения является ресурсоёмкой задачей. Так, для современного мобильного устройства, обладающего следующими характеристиками:

- разрешение экрана 1644×3840 точек;
- обновление кадра с частотой 120 Гц;

пропускная способность графического конвейера, при условии хранения цвета одного пикселя в четырёх байтах, должна составлять $1644 * 3840 * 4 * 120 = 3030220800$ байт в секунду, что составляет около 2.8 гигабайт. При условии необходимости обновления кадров с частотой 120 Гц, каждый следующий кадр должен быть подготовлен к отрисовке не позднее, чем через 8 мс после отрисовки предыдущего. Подобная нагрузка на мобильное устройство может негативно сказаться на пользовательском опыте: уменьшение времени работы устройства от одного заряда аккумулятора, потеря плавности смены кадров ввиду недостаточной производительности устройства.

По сравнению с алгоритмом, представленным в пункте 2.2.3, использование типовой информации о графических компонентах позволяет сократить количество операций, необходимых для обновления кадра.

Такая возможность, однако, подразумевает тесную интеграцию компилятора языка разработки приложений, имеющего эту информацию, в процесс отрисовки интерфейса.

Реактивные обновления пользовательского интерфейса

Реактивное программирование — парадигма программирования, ориентированная на потоки данных и распространение изменений. Это означает существование возможности выражения статических и динамических потоков данных, а также то, что нижележащая модель исполнения должна автоматически распространять изменения в рамках определённого потока данных. В контексте разработки графического интерфейса, под реактивностью понимается автоматическое обновление пользовательского интерфейса при изменении данных, помеченных каким-либо образом.

3.2. Нефункциональные требования

Декларативность описания пользовательского интерфейса мобильного приложения

В отличие от императивного программирования интерфейсов, в котором разработчик задаёт как именно необходимо построить интерфейс, декларативное описание интерфейсов подразумевает от разработчика указание того, что он хочет увидеть на экране вне зависимости то того, каким образом интерфейс будет построен.

Предоставление отладочных возможностей

Отладка программ — неизбежный процесс, возникающий при разработке большого и сложного программного обеспечения. Существуют несколько видов отладки: отладочный вывод, трассировка, использование отладчика. Наиболее совершенным способом отладки является применение специальных инструментов — отладчиков. Однако, несмотря на это, существуют ситуации, когда использование отладчика невоз-

можно по каким-либо причинам. Исходя из этого, современный язык разработки мобильных приложений должен предоставлять пользователям различные виды отладки для решения разного спектра проблем.

Кроссплатформенная разработка

Возможность разработки и поддержки единой кодовой базы мобильного приложения уже многие годы является потребностью разработчиков. Такая возможность сказывается на стоимости разработки приложения, что, в свою очередь, через стоимость конечного продукта влияет и на пользовательский опыт, получаемый потребителем от взаимодействия с приложением.

Под кроссплатформенностью понимается возможность использования единой кодовой базы приложения для его компиляции под различные целевые платформы, например, Android, iOS, без необходимости какого-либо изменения бизнес-логики или структуры графического интерфейса приложения.

Поддержка интегрированной средой разработки

Интегрированная среда разработки стала неотъемлемой частью процесса создания комплексных программных систем. Мобильные приложения являются примерами таких систем, поскольку зачастую при их разработке требуется не только окружение для работы с исходным кодом, но и для работы с ресурсами приложения, базой данных и так далее.

3.3. Соответствие существующих решений собранным требованиям

В таблице 1 отображено соответствие популярных существующих решений собранным требованиям к языку разработки мобильных приложений и его компилятору. Как видно из таблицы, ни одно из существующих решений не соответствует всем собранным в данной работе

требованиям.

	Dart / Flutter	Kotlin DSL / Jetpack Compose	JavaScript / React Native	Swift / SwiftUI
Оптимизация отрисовки графического интерфейса	−	−	−	+
Реактивные обновления пользовательского интерфейса	+	+	+	+
Декларативность описания пользовательского интерфейса	+	+	+	+
Кроссплатформенная разработка	+	+	+	−
Предоставление отладочных возможностей	+	+	+	+
Поддержка интегрированной средой разработки	+	+	+	+

Таблица 1: Таблица соответствия существующих решений собранным требованиям

4. Архитектура и особенности реализации

В данном разделе представлены архитектурные решения разрабатываемого языка и и некоторые особенности их реализации.

4.1. Архитектурные решения

В ходе данной работы, выбор тех или иных архитектурных решений был мотивирован удовлетворением итогового результата работы всем собранным в главе 3 требованиям к спецификации и компилятору современного языка разработки мобильных приложений.

4.1.1. Встраивание предметного языка в язык программирования Accord

Язык разработки мобильных приложений является примером предметно-ориентированного языка. В качестве способа реализации такого языка был выбран метод встраивания предметного языка в базовый, которым является язык Accord. Такое решение позволило переиспользовать существующий прототип компилятора и спецификацию языка Accord, включая синтаксис, синтаксический и семантический анализаторы, оптимизатор высокоуровневого промежуточного представления, кодогенерации в LLVM IR и байткод некоторой виртуальной машины.

Следствием этого решения является тот факт, что все синтаксические и семантические конструкции и правила являются общими как для разработки мобильных интерфейсов, так и для программ общего назначения. Другим следствием является автоматическое соответствие решения таким требованиям, как:

- кроссплатформенная разработка: наличие кодогенерации в LLVM IR означает возможность потенциального запуска мобильных приложений, написанных на языке Accord, на всех платформах, поддерживаемых проектом LLVM [13], а также на платформах, поддерживаемых виртуальной машиной, в байткод которой может быть оттранслирован исходный код приложения;

- предоставление отладочных возможностей: тот факт, что язык разработки мобильных приложений встроен в язык Accord, что означает их единство, гарантирует предоставление отладочных возможностей при разработке мобильных приложений в случае, если язык Accord имеет эти возможности. Несмотря на раннюю стадию разработки, язык Accord уже способен сохранять отладочную информацию о программе в формате DWARF [3];

- поддержка интегрированной средой разработки: любая интегрированная среда разработки для языка Accord подходит и для разработки мобильных приложений на нём в силу единства языка разработки мобильных приложения и языка Accord.

Для достижения декларативности описания графического интерфейса в спецификацию языка Accord были добавлены процедуры инициализации объектов. На листинге 6 представлено упрощённое синтаксическое правило определения процедур инициализации внутри синтаксического контекста определения типа. Вызов процедуры инициализации происходит автоматически при создании объекта во время выполнения программы. Синтаксически данная семантика выглядит следующим образом: *TypeName(args)*, например, *Text("This is an example")*.

```
1 INIT
2   : 'init'
3   ;
4
5 type_init
6   : INIT fn_params expression_sequence
7   ;
```

Листинг 6: Синтаксическое правило процедур инициализации

Для работы с реактивными данным в спецификацию языка Accord был добавлен модификатор полей типов — *rx*. На данный момент, для данных графической компоненты, помеченных модификатором *rx*, компилятор автоматически генерирует функцию-мутатор. Задачей данной функции является изменение реактивных данных и отметка необходимости обновления графической компоненты, реактивные данные которой были изменены. В ходе анализа графа потока управления, все

изменения реактивных данных заменяются на вызов сгенерированной функции-мутатора. На листинге 7 представлен пример мутатора реактивного поля *counter* типа *i32* компоненты *CounterComponent*.

```
fn CounterComponent.aco.set_counter(val: i32) {
    counter = val
    markNeedUpdate(this)
}
```

Листинг 7: Пример функции-мутатора реактивных данных

В дальнейшем семантика модификатора *rx* будет расширена: он будет применим не только к полям типов, но и к переменным. Изменение реактивных данных будет приводить не только к обновлению графической комопненты на экране, но и к обновлению других данных, каклибо использующих изменившиеся реактивные данные.

4.1.2. Статическая типизация графических компонент

Различные графические компоненты могут сильно отличаться друг друга семантически. Так, одни компоненты имеют динамическую природу и могут изменяться от кадра к кадру, другие же — статические — создаются лишь раз и не меняются на протяжении всей работы приложения. Реальные графические компоненты, определяемые пользователем в приложении могут быть достаточно сложными: включать большое количество различных компонент, каждая из которых имеет своё собственное поведение и условия обновления. Классический алгоритм обновления кадра, описанный в главе 2.2.3, при условии поддержки компилятором, использует статическую информацию о компонентах для минимизации количества действий, необходимых для обновления кадра. Так, информация о том, может ли компонента изменяться во время работы приложения, используется средой времени исполнения для уменьшения количества сравнений соответствующих компонент двух соседних кадров. Однако, знание таких параметров, как тип и размер компоненты и её подкомпонент, а также расположение подкомпонент относительно их родительской компоненты позволяет избавиться от части операций, проводимых классическим алгоритмом во время испол-

нения приложения ещё на этапе компиляции программы.

Рассмотрим пример графической компоненты на листинге 8. Она представляет собой колонку, состоящую из текста, кнопки и некоторой условной компоненты, которая превращается в изображение или текст в зависимости от условия *condition*.

```
Column {
    Text(''Current count: ${counter}'')
    Button(''Click on me!'')
        .onClick(fn() { counter += 1 })
        .backgroundColor(Color.Green)
        .width(150px)
    ConditionalView(
        condition,
        Image(''img.png''),
        Text(''Empty'')
    )
}
```

Листинг 8: Пример графической компоненты

Зная статический тип данной компоненты (листинг 9: $f0$ — тип подкомпоненты-колонки, $f1$ — тип переменной *counter*, $f2$ — тип переменной *condition*) и её представление в памяти (рис. 2), компилятор способен ещё во время компиляции приложения понять, какие компоненты могут быть изменены во время работы приложения, а какие нет.

```
struct {
    f0: Column[Text, Button, ConditionalView[Image, Text]]
    f1: i32
    f2: bool
}
```

Листинг 9: Пример статического типа компоненты

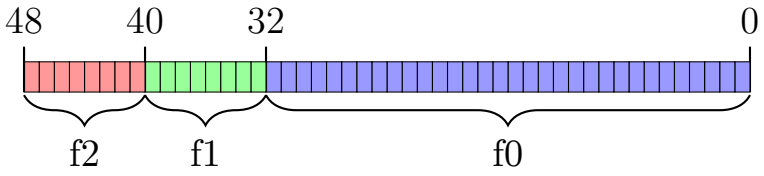

Рис. 2: Пример представления компоненты в памяти

Имея представление компоненты в памяти, компилятор может сгенерировать специализированную для конкретного типа компоненты про-

цедуру обновления (листинг 10). Эта процедура состоит из точечных вызовов обновления только потенциально изменяемых во время работы приложения компонент. Для сравнения такого подхода с классическим, введём понятия некоторых условных операций, необходимых для обновления интерфейса согласно главе 2.2.3. Пусть A — операция перехода между узлами дерева компонент, B — проверка узла на изменяемость (проверка во время исполнения программы), C — вызов процедуры обновления компоненты. Тогда, если N — количество всех компонент, M — количество изменяемых компонент, причём $M \leq N$, то для обновления одного кадра классическому алгоритму необходимо произвести $(N-1) * A + N * B + M * C$ операций, в то время как описанному выше алгоритму лишь $M * C$.

```
fn Counter.rerender() {
    // skip Column
    Text.rerender(fieldAddress(0, 0), args...)
    // skip Button
    ConditionalView.rerender(
        fieldAddress(0, 2), args...,
    )
}
```

Листинг 10: Пример сгенерированной процедуры обновления компоненты

Для того, чтобы данная информация была доступна компилятору языка Accord, в его спецификацию были добавлены синтаксис и семантика наследования независимых типов (не имеющих типов-параметров) от ненастроенных обобщённых типов с последующей автоматической настройкой обобщённого типа-родителя в зависимости от содержимого определения типа-потомка.

4.2. Вывод

Результатом выбора и реализации архитектурных решений, описанных в пункте 4.1, стало соответствие разработанного решения всем требованиям, перечисленным в главе 3.

5. Апробация

Предложенные и описанные в пункте 4.1 синтаксические и семантические изменения в спецификации языка Accord были успешно реализованы в прототипе компилятора языка. На листинге 11 представлен пример графической компоненты, описанной на изменённом языке Accord. Данный пример успешно компилируется прототипом компилятора в LLVM IR и байткод некоторой виртуальной машины. На данный момент компиляция полноценных приложений, использующих такие графические компоненты, невозможна ввиду отсутствия интегрированного с компилятором движка отрисовки. Эта интеграция является одной из следующих задач в рамках развития направления разработки мобильных приложений на языке Accord.

```
type Counter = struct (View[T]) {
    rx var counter: i32
    var condition: bool

    var body: T = Column(
        Text(''Current count: ${counter}''),
        Button(''Click on me!'')
            .onClick(fn() { counter += 1 })
            .backgroundColor(Color.Green)
        ConditionalView(
            condition,
            Image(''img.png''),
            Text(''No image'')
        )
    )
}
```

Листинг 11: Пример графической компоненты на языке Accord

6. Заключение

В ходе выполнения данной выпускной квалификационной работы были достигнуты следующие результаты:

- выполнен обзор предметной области:
  - предметно-ориентированные языки;
  - отрисовка графического интерфейса;
- выполнен обзор существующих решений:
  - *Dart / Flutter*;
  - *Kotlin DSL / Jetpack Compose*;
  - *JavaScript / React Native*;
  - *Swift / SwiftUI*;
- собраны и проанализированы требования к современному языку разработки мобильных приложений и его компилятору;
- предложены изменения спецификации и компилятора языка Accord, позволяющие удовлетворить всем собранным требованиям;
- предложенные изменения реализованы в прототипе компилятора языка Accord;
- пример графической компоненты успешно проходит компиляцию прототипом компилятора языка Accord.

Список литературы

[1] ANTLR Homepage. — Access mode: https://www.antlr.org/ (online; accessed: 18.04.2021).

[2] Advaitha B Mopuru Lahari Gopalakrishnan T. Trends in Processor Architecture of Mobile Phones: A Survey // International Journal of Advanced Science and Technology. — 2020. — May. — Vol. 29, no. 05. — P. 6265 – 6274. — Access mode: http://sersc.org/journals/index.php/IJAST/article/view/15631.

[3] DWARF Debugging Information Standard Homepage. — Access mode: http://dwarfstd.org/ (online; accessed: 28.04.2021).

[4] Dart Homepage. — Access mode: https://dart.dev/ (online; accessed: 10.04.2021).

[5] Flutter Homepage. — Access mode: https://flutter.dev/ (online; accessed: 02.12.2020).

[6] Go Homepage. — Access mode: https://golang.org/ (online; accessed: 28.04.2021).

[7] Gray Jeff, Karsai Gabor. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. — 2003. — 01. — P. 325.

[8] Jetpack Compose Homepage. — Access mode: https://developer.android.com/jetpack/compose (online; accessed: 02.05.2021).

[9] Kelly Steven, Tolvanen Juha-pekka. Domain-Specific Modeling: Enabling Full Code Generation. — 2008. — 04. — ISBN: 978-0-470-03666-2.

[10] Kolawole Emmanuel Olawale, Lofinmakin Damilola Ayomiposi, Nwidobie Gabriel. Trends in Mobile Phones Processor Architecture, Academia. — Access mode: https://www.academia.edu/38755927/

Trends_in_Mobile_Phones_Processor_Architecture (online; accessed: 02.12.2020).

[11] Kotlin Homepage. — Access mode: https://kotlinlang.org/ (online; accessed: 02.12.2020).

[12] Kramer D., Clark T., Oussena S. MobDSL: A Domain Specific Language for multiple mobile platform deployment // 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications. — 2010. — P. 1–7.

[13] LLVM Homepage. — Access mode: https://llvm.org/ (online; accessed: 28.04.2021).

[14] LLVM IR Language Reference. — Access mode: https://llvm.org/docs/LangRef.html (online; accessed: 28.04.2021).

[15] McCracken Daniel D., Reilly Edwin D. Backus-Naur Form (BNF) // Encyclopedia of Computer Science. — GBR : John Wiley and Sons Ltd., 2003. — P. 129–131. — ISBN: 0470864125.

[16] Mernik Marjan, Heering Jan, Sloane Anthony. When and How to Develop Domain-Specific Languages // ACM Comput. Surv. — 2005. — 12. — Vol. 37. — P. 316–.

[17] Mobile Applications Download Statistics. — Access mode: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/ (online; accessed: 02.12.2020).

[18] Mobile Devices Pixel Density Statistics. — Access mode: https://pixensity.com/list/phone/ (online; accessed: 20.12.2020).

[19] Mobile Devices Sales Statistics. — Access mode: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/ (online; accessed: 02.12.2020).

[20] React Native Homepage. — Access mode: https://reactnative.dev/ (online; accessed: 02.12.2020).

[21] Sippu Seppo, Soisalon-Soininen Eljas. On LL (k) parsing // Information and Control. — 1982. — Vol. 53, no. 3. — P. 141–164.

[22] Spinellis Diomidis. Notable design patterns for domain-specific languages // Journal of Systems and Software. — 2001. — Vol. 56, no. 1. — P. 91 – 99. — Access mode: http://www.sciencedirect.com/science/article/pii/S0164121200000893.

[23] Swift HomePage. — Access mode: https://developer.apple.com/swift/ (online; accessed: 03.04.2021).

[24] Swift Protocols Documentation. — Access mode: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (online; accessed: 18.04.2021).

[25] SwiftUI HomePage. — Access mode: https://developer.apple.com/xcode/swiftui/ (online; accessed: 03.04.2021).

[26] Vue Native HomePage. — Access mode: https://vue-native.io/ (online; accessed: 02.12.2020).

[27] Wile D. S. Supporting the DSL Spectrum // Journal of computing and information technology. — 2001. — Vol. 9, no. 4. — P. 263 – 287.

[28] A preliminary study on various implementation approaches of domain-specific language / Tomaž Kosar, Pablo E. Martı´nez López, Pablo A. Barrientos, Marjan Mernik // Information and Software Technology. — 2008. — Vol. 50, no. 5. — P. 390 – 405. — Access mode: http://www.sciencedirect.com/science/article/pii/S0950584907000419.