МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет Механико-математический

Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Кареба Юрий Сергеевич

(Фамилия, Имя, Отчество автора)

Тема работы:

Реализация мультипарадигменного промежуточного представления оптимизирующего компилятора

«К защите допущена»

Заведующий кафедрой

к.ф.-м.н.

Емельянов П.Г./……

(фамилия , И., О.) / (подпись, МП)

«……»……2026г.

Научный руководитель

Старший преподаватель

ММФ НГУ

Трепаков И.С./……

(фамилия , И., О.) / (подпись, МП)

«……»……2026г.

Дата защиты: «……»……2026г.

Новосибирск, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Скорость работы оптимизатора, эффективность порождаемого им кода, удобство его разработки, во многом зависят от выбора промежуточного представления. Исходный язык программирования определяет набор машинно-независимых оптимизаций и частоту их применения, следовательно влияет на выбор промежуточного представления.

В контексте исследования зависимости промежуточного представления от исходного языка, удобно категоризировать языки программирования (и соответственно их промежуточные представления) по наборам абстракций, которые они предоставляют. В настоящей работе мы будем пользоваться известной классификацией языков на функциональные и императивные. К последним также будем относить и современные объектно–ориентированные языки в силу схожести устройства их компиляторов.

Языки программирования разных парадигм долгое время развивались независимо друг от друга. Поэтому их промежуточные представления имеют различия. Компиляторы императивных языков активно используют графовые промежуточные представления с явным потоком управления. Функциональные компиляторы естественным образом представляют функции в промежуточном представлении.

В настоящее время языки программирования все более тяготеют к смешению парадигм: абстракции, изначально появившиеся в функциональных языках, перетекают и активно используются в

объектно-ориентированных языках. Самым ярким примером такого заимствования являются функциональные замыкания, которые можно найти в любом современном императивном языке. Но промежуточные представления после подобных заимствований часто остаются неизменными, и заимствованные концепции выражаются в терминах исходного языка [1–3], что приводит к ухудшению результатов оптимизаций.

Отсюда возникает идея реализации *мультипарадигменного промежуточного представления*. В этой работе содержатся результаты разработки функционального расширения императивного промежуточного представления и анализ его эффективности в рамках статического оптимизирующего компилятора многоязыковой виртуальной машины Huawei.

1 Предметная область

1.1 Промежуточные представления языков программирования

Парадигма языка программирования напрямую влияет на промежуточное представление компилятора этого языка.

Основными абстракциями императивных языков программирования являются последовательное исполнение выражений, использование управляющих конструкций, наличие переменных, разбиение программ на процедуры или функции. Управляющие конструкции делают поток исполнения программы нетривиальным, чтобы представлять все возможные пути исполнения и анализировать поведения программы вдоль них используется классическое промежуточное представление программы в виде *графа потока управления* (англ. *Control Flow Graph*, *CFG*) [4]. Для упрощения анализа потока данных и оптимизаций вместе с CFG используется SSA-форма программы [5]. Наследником идей этих представлений является *Sea of Nodes*, которое используется в современных статических и динамических компиляторах для объектно-ориентированных языков программирования [6,7].

Основными абстракциями функциональных языков программирования являются функциональные замыкания, функции высших порядков, полиморфные функции, алгебраические типы данных, классы типов. Языки этой парадигмы, такие как Haskell, SML, Scheme, выбирают в качестве промежуточного представления различные

модификации лямбда-исчисления: Административную нормальную форму [8] или Continuation Passing Style [9,10].

В следующих разделах представлен обзор наиболее популярных промежуточных представлений.

1.2 Императивные промежуточные представления

1.2.1 Граф потока управления

Граф потока управления (англ. *Control Flow Graph*, *CFG*) – это ориентированный граф, узлы которого называются *базовыми блоками*. Дуги этого графа обозначают *передачу потока управления*. Граф потока управления содержит две выделенные вершины – *стартовый блок*, не имеющий предшественников, и *выходной блок*, не имеющий потомков. Каждый базовый блок содержит в себе набор инструкций.

Граф потока управления, как и предполагается из названия, явно содержит в себе информацию о всех возможных путях исполнения программы. То есть всякому пути исполнения исходной программы соответствует определенный путь в графе потока управления.

Инструкции в блоке выполняются последовательно, последняя инструкция базового блока определяет какому из потомков блока будет передано управление. Такие инструкции обычно соответствуют управляющим конструкциям исходного языка: инструкция `goto` – безусловный переход к единственному наследнику текущего блока, условный переход `if` определяющий передачу управления по логическому

значению предиката, условный выбор одного из многих потомков – `switch` и т.д.

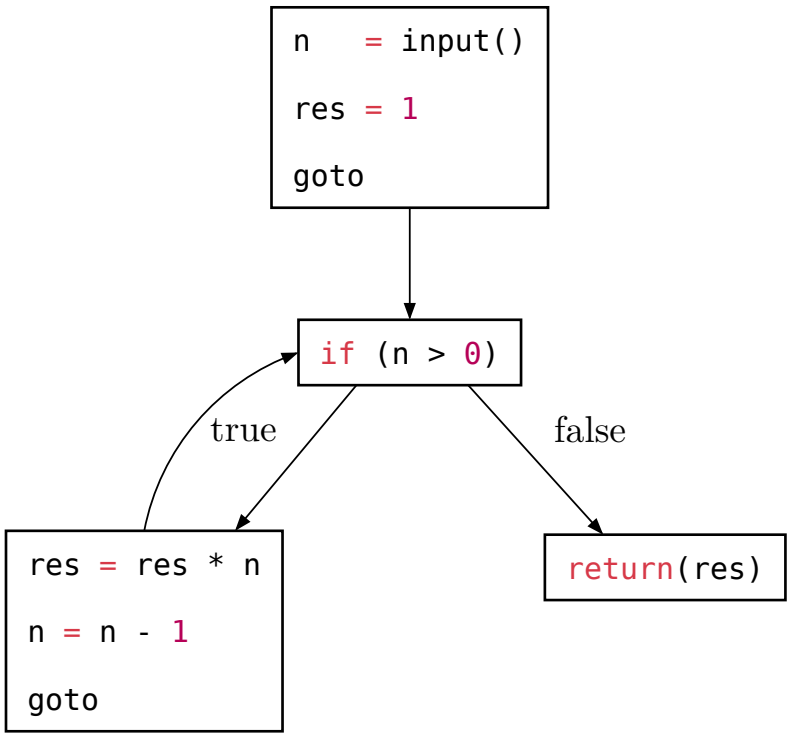


Рисунок 1 — Пример CFG для функции вычисляющей факториал

Явное присутствие потока управления в промежуточном представлении позволяет судить о свойствах программы в зависимости от пути её исполнения. Такая техника анализа программы называется *потоковым анализом* (англ. *Data-flow analysis*, *DFA*)[11]. Информация, полученная в ходе потокового анализа, используется компилятором для проведения различных оптимизаций [12].

1.2.2 SSA

До текущего момента мы не говорили о том в каком виде инструкции представлены внутри базового блока, предполагая, что они точно такие же, как и в исходном коде программы. Но такое «естественное» представление инструкций неудобно для оптимизаций. В

частности потому, что исходный код может содержать переопределения переменных, что мешает компилятору понять какое именно определение переменной используется в инструкции. Эту информацию приходится перевычислять в ходе анализов и оптимизаций или хранить в специальных структурах данных, называемых *def-use* цепями.

def-use цепью будем называть структуру данных, которая для каждой инструкции в CFG, объявляющей переменную, содержит список всех *определений*, использованных в инструкции и список всех использований объявленной переменной.

Недостаток def-use цепей заключается в стоимости их использования: для переменной имеющей N использований и M объявлений, размер def-use цепей пропорционален $N \cdot M$. Но существует и другое решение, более компактное в большинстве практических случаев и – *Single Static Assignment* (сокращенно SSA) форма программы [13,14].

Определение 1.2.1. Будем говорить, что программа находится в *SSA* форме, если любая переменная определяется ровно один раз и не может быть переопределена.

Линейный участок кода нетрудно переписать в SSA-форму, достаточно заметить, что вместо инструкций переопределяющих переменную, можно использовать инструкции определяющие новую переменную. Вместо переопределения переменной `a` мы будем создавать определение новой версии этой переменной `a_i`, как показано на листинге 1.

```scala
def simple_ssa() = {
    var a = 0
    var b = 3
    a = 16 - 14
    b = 3 * 7
    a = a * b
    a
}
```

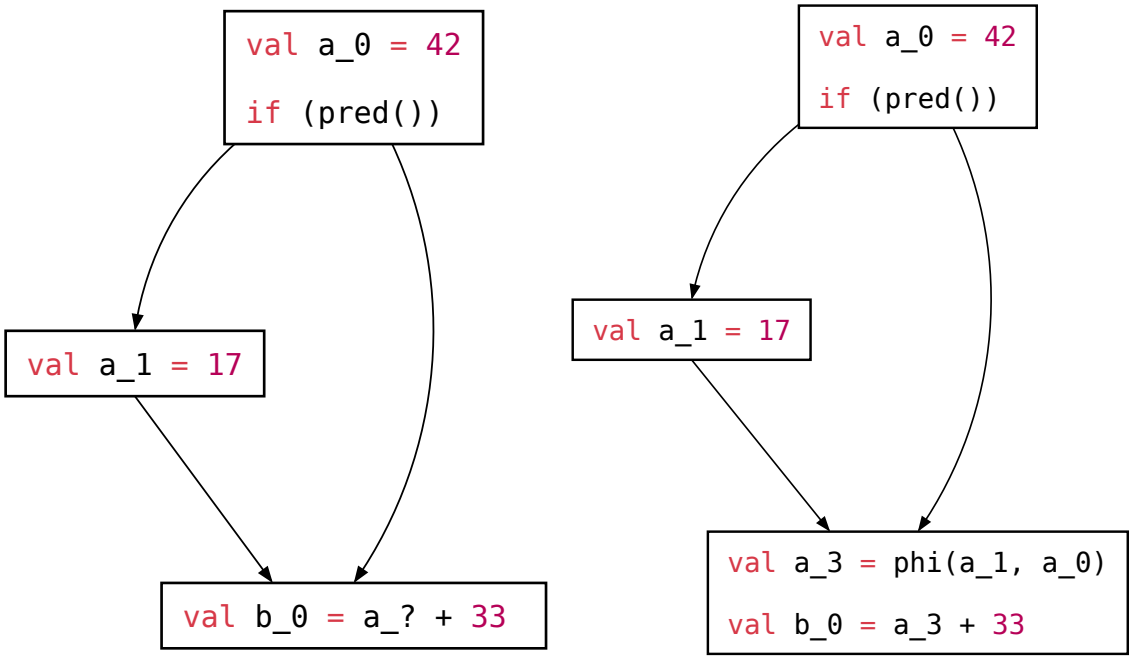
(а)

```scala
def simple_ssa() = {
    val a_0 = 0
    val b_0 = 3
    val a_1 = 16 - 14
    val b_1 = 3 * 7
    val a_2 = a_1 * b_1
    a_2
}
```

(б)

Листинг 1 — (а) – программа на языке Scala, (б) – программа после построения SSA-формы

Проблемой становится выбор версии переменной при схождении потока управления в программе. Рассмотрим CFG, изображенный на рисунке 2(а). Версия переменной `a`, достигающая определение переменной `b_0`, зависит от пути исполнения программы. В таком случае неясно какую версию переменной использовать для преобразования программы в SSA-форму, выбор любой из двух версий будет некорректным. Для решения этой проблемы в SSA-форме вводится особая инструкция - φ-функция.



(а) (б)

Рисунок 2 — (а) – участок CFG со схождением управления (б) – применение φ-функции в точке схождения управления

φ-функции определяют версию переменной в точках схождения управления в CFG. φ-функция принимает в качестве аргументов версии переменной, а её результатом является одна из версий, выбранная в соответствии с потоком управления. Обратимся к рисунку 2(б), значение φ-функции будет равно значению `a_1`, если управление будет передано из блока, содержащего `a_1`, в случае если управление будет передано из входного блока значение, то значение φ-функции будет равно значению `a_0`.

Определение 1.2.2 (SSA-граф). Вершинами ориентированного SSA-графа являются объявления SSA-переменных, дуга между вершинами v и u означает, что инструкция, объявляющая u использует значение переменной v.

В SSA-форме зависимости между объявлениями и использованиями переменных выражаются в виде SSA-графа.

Для всякого графа потока управления можно построить SSA–формы, использующие разное количество φ–функций. Существуют классические алгоритмы построения SSA [13,14], использующие в некотором смысле минимальное количество φ–функций.

SSA–форма программы тесно связана с отношением доминирования:

Свойство 1.2.3 (Отношение доминирования). Инструкция графа потока управления u доминирует инструкцию v если и только если на любом пути из входного блока до v встречается u.

Обозначается $u \operatorname{dom} v$. Отношение доминирования является рефлексивным транзитивным отношением.

Свойство 1.2.4 (Доминирование SSA–переменных). Если переменные CFG находятся в SSA-форме, то любое использование переменной доминируется её определением

1.2.3 Sea of Nodes

Инструкции имеют закрепленное положение внутри графа потока управления. Первоначальные местоположения инструкций определяются исходной программой и, как следствие, могут быть не оптимальными с точки зрения производительности программы.

Наиболее яркий пример неоптимального размещения инструкций – инвариантные вычисления в цикле. Код цикла исполняется часто и его эффективность оказывает сильное влияние на производительность. Поэтому выгодно оставлять в цикле как можно меньше вычислений. Инвариантные вычисления могли оказаться в цикле по вине программиста или могли появиться в ходе оптимизации программы.

```
1 val n = 1_000_000
2 val arr = Array.fill[Int](n)(0)
3 val (a, b) = (17, 42)
4 val arr = Array.fill(0)
5 for (i <- 0 to n) {
6   val t = (a % b) * b * a % 97
7   arr(i) = i + t;
8 }
```

Листинг 2 — Цикл с избыточными вычислениями

В шестой строке листинга 2 содержатся вычисления, не использующие индуктивную переменную цикла. Поэтому определение переменной `t` можно вынести из тела цикла. Если этого не сделать, то `t` будет перевычисляться на каждой из 1000000 итераций.

Также программа может содержать частично избыточные вычисления, для примера рассмотрим программу на листинге 3. Определение переменной `t` в шестой строке вычисляется каждую итерацию цикла. Но `t` имеет единственное использование только в `true` ветке условного перехода на строке 7, которая выполнится всего лишь 10 раз. Если перенести вычисление к использованию, то количество вычислений внутри цикла уменьшится.

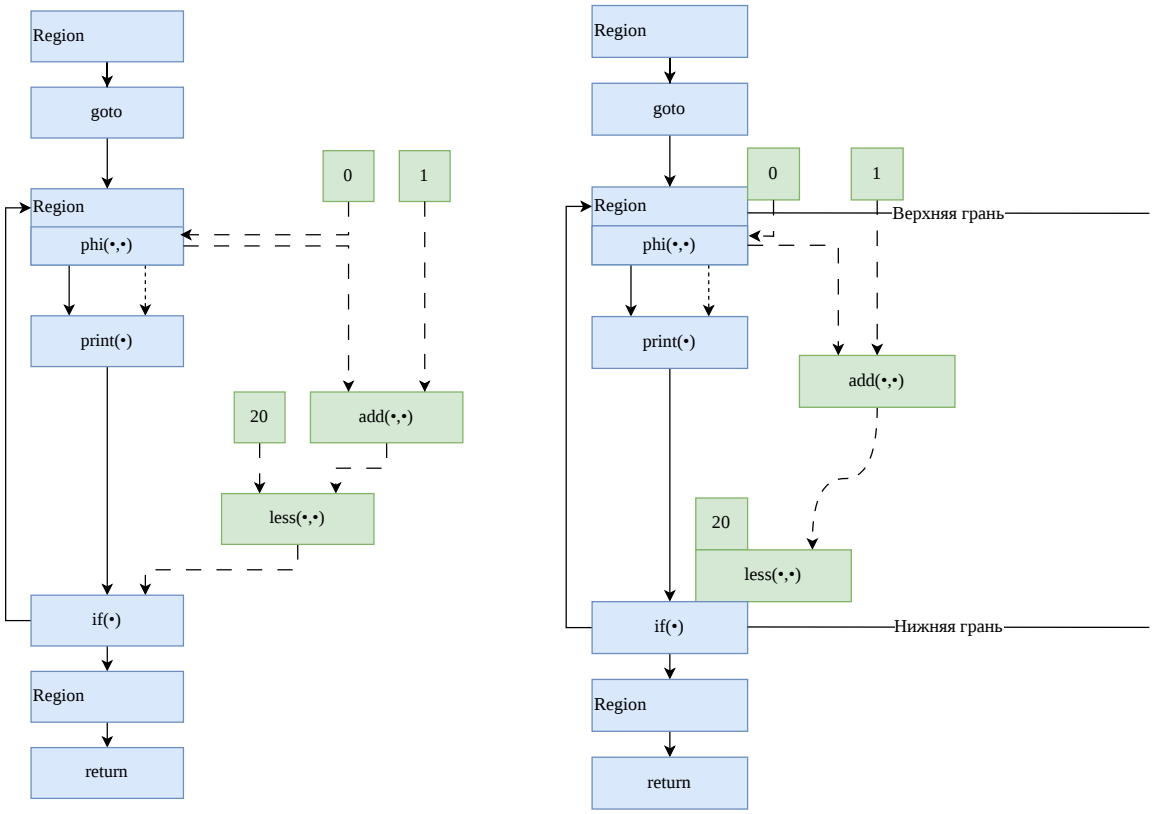
```
val n = 1_000_000
val arr = Array.fill[Int](n)(0)
val (a, b) = (17, 42)
val arr = Array.fill(0)
for (i <- 0 to n) {
  val t = (a % b) * i * a % 97
  if (i % 100_000) {
    arr(i) = i + t
  } else {
    arr(i) = i + 42
  }
}
```

Листинг 3 — Цикл с частично избыточными вычислениями

Анализы и трансформации, перемещающие инструкции внутри CFG, должны одновременно работать с двумя структурами данных – CFG и SSA-графом. CFG содержит информацию о потоке управления, SSA-граф представляет зависимости между данными программы. Это усложняет реализацию описанных выше оптимизаций. Более того, Клифф Клик показал, что такая архитектура препятствует комбинированию различных оптимизационных техник.

В диссертации Клика и ряде сопутствующих статей [6,15] было представлено графовое промежуточное представление *Sea of Nodes*. В Sea of Nodes есть закрепленные инструкции, эти инструкции не могут быть перемещены и как правило, они формируют линейные участки Sea of Nodes графа. Закрепленными инструкциями, например, являются начала линейных участков `Region`, инструкции безусловных и условных переходов, φ–функции. Часть инструкций промежуточного не имеет закрепленного положения, такие инструкции называются плавающими.

На рисунке 3(а) показан пример промежуточного представления Sea of Nodes.



(а) (б)

Рисунок 3 — Пример Sea of Nodes графа синим цветом отмечены закрепленные инструкции, зеленым – плавающие.

На рисунке (б) отмечены верхняя и нижняя грания для инструкции `add`.

Однако, инструкции целевой архитектуры не могут «плавать» в машинном коде, они должны быть сгенерированы в определенном месте. Для того чтобы провести кодогенерацию, а также для проведения структурных трансформаций, компилятор должен выбрать подходящее место для плавающих инструкций. Процесс размешения инструкций Sea of Nodes графа называется *линеаризацией* или *планированием*, а алгоритм, выбирающий наиболее подходящие места для инструкций, называется *Global Code Motion* (сокращенно *GCM*)[15].

Алгоритм планирования инструкций не может разместить плавающую инструкцию в произвольной точке, он должен учитывать зависимости между инструкциями. Для каждой инструкции компилятор вычисляет верхнюю и нижнюю грани. Инструкция может быть безопасно размещена только между этими границами.

Определение 1.2.5 (Верхняя грань плавающей инструкции). *Верхней гранью* плавающей инструкции `v`, расположение аргументов которой уже определено, называется закрепленная инструкция, являющаяся наименее глубокой в дереве доминаторов вершиной, доминируемой всеми аргументами `v`.

Определение 1.2.6 (Нижняя грань плавающей инструкции). *Нижней гранью* плавающей инструкции `v`, расположение использований которой уже определено, называется наименьший общий предок использований `v` в дереве доминаторов [4] Sea of Nodes графа.

Место в промежутке между нижней и верхней гранью инструкции определяется согласно эвристикам. Например, алгоритмы планирования инструкций стараются вынести из цикла как можно больше инвариантных вычислений.

Алгоритм планирования инструкций может тесно взаимодействовать с последующими стадиями компиляции [16]. Например, для улучшения качества распределения регистров алгоритмы GCM могут учитывать давление на регистры, выполнять

рематериализацию (использовать копии вычисления, если это выгодно), собирать информацию, полезную для распределения регистров.

1.3 Функциональные промежуточные представления

1.3.1 Лямбда-исчисление

Лямбда-исчисление – это формальная система, основанная на функциях. Создание функции исчисления происходит при помощи *лямбда-абстракции*, процесса связывания имен переменных тела функции с её формальными параметрами. Абстракция обозначается как λx.y, где x – формальный параметр функции, а t – это терм исчисления, являющийся телом созданной функции. Все вхождения переменной x, содержащиеся в теле абстракции t, называются *связанными* (или можно сказать, что λx. – это связывающее определение с областью видимости t). Переменные, которые не являются связанными, будем называть *свободными*. Терм исчисления, не содержащий свободных переменных, будем называть *замкнутым*. К примеру функция λx.x является замкнутой, а λx.y нет, так как содержит свободную переменную y. Применение функции к её аргументам – это подстановка аргументов функции в её тело на место формальных параметров. Определение лямбда-исчисления, а также формальное определение функции подстановки на термах исчисления можно найти в книге «Типы в языках программирования» [17].

Программировать на классическом лямбда-исчислении довольно сложно, так как оно не содержит ничего кроме функций. Поэтому

в функциональные языки добавляют различные синтаксические конструкции, типизацию. Вслед за языками эти расширения лямбда исчисления перетекают и в промежуточные представления.

Для демонстрации различных функциональных промежуточных представлений будем использовать простой язык с ml-подобным синтаксисом[1]. В языке доступны `let-in` и `if-then-else` выражения, типизация и алгебраические типы данных, сопоставление с образцом при помощи `case-of` выражений. Пример использования этого промежуточного представления можно найти на листинге 4.

```
1 fun fact n =
2   if n = 0 then 1
3   else n * fact (n - 1)
```

Листинг 4 — Пример языка промежуточного представления, вычисляющий факториал числа `n`

К построению промежуточных представлений оптимизирующих компиляторов функциональных языков существует два глобальных подхода, это *стиль передачи продолжений* и *прямой стиль*. В следующих разделах мы коротко обсудим каждый из них.

1.3.2 Стиль передачи продолжений

Будем говорить, что программа удовлетворяет стилю передачи продолжений (англ. Continuation Passing Style, CPS), если каждая функция этой программы заканчивается хвостовым вызовом, то есть

[1]MetaLanguage – это функциональный язык программирования, разработанный в конце 1970х. Его последователи (Caml, SML) обладают узнаваемым синтаксисом, который похож на синтаксис на лямбда-исчисления, но более читаем.

последним действием каждой функции является вызов другой функции. Стиль передачи продолжений делает поток управления функциональной программы явным, упрощая некоторые оптимизации.

Например, программа `foo` на листинге 5 не удовлетворяет стилю передачи продолжений, так как функции `t`, `f` не завершаются вызовом функции, также как и объемлющая функция `foo`.

```
1 fun foo x =
2   let fun t x = x
3   in let fun f x = x + 1
4   in let res = if x mod 2 = 0 then t x else f x
5   in 2 * res
```

Листинг 5 — Промежуточное представление, не удовлетворяющее стилю передачи продолжений.

Однако, эту программу можно привести к стилю передачи продолжений, добавив в неё продолжения. Продолжение – это функция, которая принимает результат вычисления и определяет дальнейшее исполнение. Продолжения необходимы для поддержания CPS-свойства.

Эквивалентную программу, использующую Стиль передачи продолжений можно увидеть на листинге 6.

Любую программу на функциональном языке программирования можно привести к стилю передачи продолжений за время, пропорциональное размеру программы [18,19].

```
fun foo x (ret: int -> 'a) =
  let cont x: int -> 'a = ret (2 * x)
  let fun t x (k: int -> 'a) = k x
  in let fun f x (k: int -> 'a) = k (x + 1)
  in if x mod 2 = 0 then t x cont else f x cont
```

Листинг 6 — Промежуточное представление, эквивалентное программе с листинга 5, удовлетворяющее CPS.

Стиль передачи продолжений используется на практике как промежуточное представление оптимизирующих компиляторов [10,19] и как основа для библиотек асинхронного программирования [20].

CPS хорошо подходит для трансформаций, изменяющих поток управления. Например для преобразования коммутирующих конверсий.

Рассмотрим программу, изображенную на листинге 7(а). Значение `case` выражения связывается с именем переменной `x` в большом выражении `BIG`. Выражения `let-in` и `case-of` коммутируют, то есть могут быть переставлены, как показано на листинге 7(б).

```
let val x = case v
  of p1 -> N
  of p2 -> M
in BIG
```

(а)

```
case v
  of p1 -> let val x = N
           in BIG
  of p2 -> let val x = M
           in BIG
```

(б)

Листинг 7 — (а) `let-case` выражение,

(б) Выражение после применения коммутирующей конверсии

Подобные преобразования часто используются в компиляторах функциональных языков программирования, поскольку могут открыть новые возможности для оптимизаций. Проблема проблема

коммутирующих конверсий в том, что часть выражений копируется. В примере на листинге 7 было скопировано выражение `BIG`. Вложенные `let-case`, `case-case` выражения и их коммутирующие преобразования могут привести к потенциально экспоненциальному росту размера кода [21].

Для того чтобы выполнять коммутирующие конверсии без копирования предлагается использовать продолжение `j`, получающее результат вычисления выражения `M` или `N`. Пример перестановки `let` и `case` выражений с использованием продолжений можно найти на листинге 8

```
let j x = BIG
case v
  of p1 -> let x = N in j x
  of p2 -> let x = M in j x
```

Листинг 8 — Применение коммутирующей конверсии с использованием продолжения к примеру из листинга 7(a)

Использование CPS в этом случае позволяет избежать копирования выражения `BIG`. Коммутирующие конверсии, реализованные таким образом, используются в компиляторе Glasgow Haskell Compiler [8,22].

1.3.3 Прямой стиль

Промежуточные представления прямого стиля (англ. *Direct Style*) не используют продолжения в своей основе. К языку могут применяться другие правила и ограничения. Например административная нормальная форма (сокращенно АНФ)[23] требует, чтобы каждое промежуточное вычисление было названо при помощи `let-in` выражения.

Рассмотрим программу на листинге 9(а). Программа содержит следующие промежуточные вычисления: вызов функции `g`, вызов функции `h`, вычисление суммы результатов функций и вызов функции `f`. Административную форму этой программы можно увидеть на листинге 9(б).

```
let val r = f (g x + h y)
in P
```

(а)

```
let val x1 = g x
in let val x2 = h y
in let val x3 = x1 + x2
in let val r = f x3
in P
```

(б)

Листинг 9 — (а) промежуточное представление программы, (б) эквивалентное промежуточное представление, находящееся в АНФ

Применение административной нормальной формы упрощает анализ и трансформацию потока данных в программе. Прямой стиль построения промежуточных представлений широко используется в компиляторах функциональных языков программирования, например оптимизатор Glasgow Haskell Compiler использует в качестве одного из промежуточных представлений программы язык прямого стиля *Core* [8].

1.4 Thorin IR

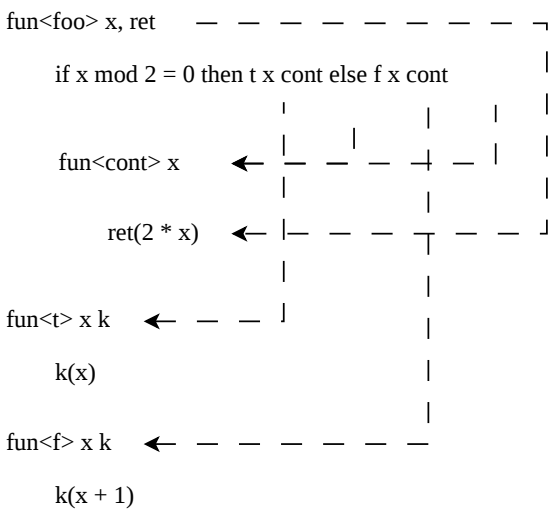
Thorin IR – это промежуточное представление частичного специализатора *AnyDSL*[24]. Thorin использует CPS в качестве основы, усиливая его сильные стороны императивными элементами.

Программирование в стиле передачи продолжений позволяет компилятору AnyDsl применять оптимизации и анализы, используемые компиляторами функциональных языков.

Thorin IR – это графовое промежуточное представление, каждая инструкция которого является вершиной графа, а зависимости между инструкциями явным образом выражены через дуги графа. Поток управления выражается через использования функций в инструкциях хвостовых вызовов. Часть инструкций являются плавающими как в Sea of Nodes.

За счет графовой структуры функции в Thorin не имеют явной вложенности. Обратимся к листингу 10, слева промежуточное представление, удовлетворяющее CPS, все функции явно вложены в `foo`, на листинге 10(б) эквивалентное Thorin представление, где вложенность функций выражена неявно через зависимости между инструкциями промежуточного представления. Например функция `cont` должна быть вложена в функцию `foo`, так как использует её параметр `ret`. Функции `t` и `f` не замыкают контекст функции `foo` поэтому могут не быть вложенными в неё.

```
fun foo x (ret: int -> 'a) =
  let cont x = ret (2 * x)
  in let fun t x k = k x
  in let fun f x k = k x + 1
  in if x mod 2 = 0
       then t x cont
       else f x cont
```



(а) (б)

Листинг 10 — (а) – Пример классического CPS-представления программы, (б) – Thorin версия этого представления

Императивные компиляторы, как правило, оптимизируют промежуточное представление одной функции. Промежуточное представление Thorin может одновременно анализировать несколько функций, что характерно для функциональных промежуточных представлений.

Смешение возможностей разных промежуточных продолжений позволяют выражать императивные оптимизации в терминах функциональных, например, расслоение и раскрутка циклов выполняется компилятором AnyDSL в терминах частичной специализации функций.

1.5 MLIR

MLIR [25] – это гибридное и расширяемое промежуточное представление, позволяющее выражать различные концепции, находясь

в рамках одной инфраструктуры. MLIR активно используется как промежуточное представление предметно-ориентированных языков программирования в сфере исскуственного интеллекта [26].

Структуру MLIR задают две сущности:

– Операции

– Регионы

Операция – основная единица промежуточного представления. При помощи операций можно выразить как сложение целых чисел `arith.addi`, так и цикл `scf.for`. Операции находятся в SSA форме.

Регионы – это набор базовых блоков. Регионы используются для моделирования вложенных вычислений и областей видимости. Регионы могут принадлежать операциям. Например операция `scf.for` содержит один регион, являющийся телом цикла. Базовые блоки MLIR содержат набор последовательных операций.

Такая структура абстрактная структура MLIR позволяет легко создавать сложные пользовательские операции.

2 Постановка задачи

Многоязыковая виртуальная машина Huawei поддерживает языки Scala, Java, Cangjie. Scala – это мультипарадигменный язык программирования, активно использующий элементы функциональной парадигмы. Java и Cangjie – императивные языки, но также используют элементы функциональной парадигмы.

Статический компилятор многоязыковой виртуальной машины Huawei использует промежуточное представление типа Sea of Nodes. На уровне промежуточного представления концепции функционального программирования, такие как анонимные функции, выражаются при помощи объектов, что усложняет некоторые оптимизации и анализы. Отсюда возникает идея добавить в промежуточное представление компилятора элементы, специфичные функциональным промежуточным представлениям, и исследовать преимущества и недостатки такого подхода.

Таким образом, цель данной работы – разработка расширения императивного SSA-представления оптимизирующего компилятора элементами функционального представления CPS и исследование сферы применимости такого расширения.

2.1 Промежуточное представление компилятора

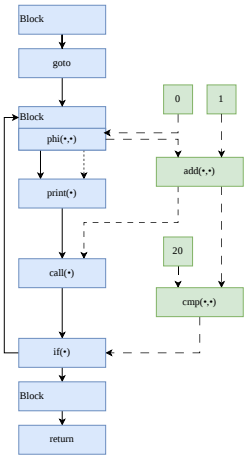
Рассмотрим промежуточное представление оптимизирующего компилятора виртуальной машины Huawei. В промежуточном представлении явно выражены линейные участки. Линейный

участок начинается с базового блока и содержит в себе линейную последовательность закрепленных инструкций, заканчивается инструкцией передачи управления. Закрепленной инструкцией, например, явялется инструкция вызова метода `Call`. Инструкции завершающие линейные участки это `Goto`, `If`, `Return`, `Switch`.

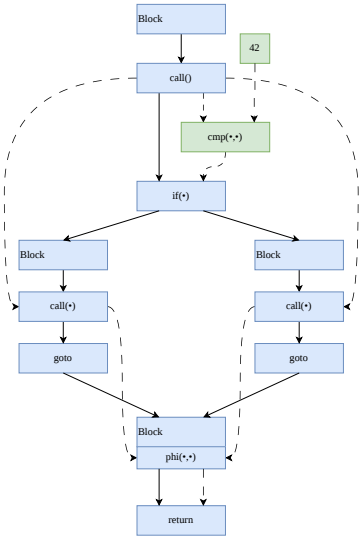
К базовому блоку прикрепляются φ-функции. Часть инструкций не имеют закрепленного положения внутри линейных участков, будем называть их плавающими. Плавающими инструкциями, например, являются константы, инструкции сложения.

На рисунке 11 изображены примеры промежуточного представления.

```
var i = 0
do {
  print(i)
  i += 1
  call(i)
} while (i < 20)
```




```
val a = foo()
var b;
if (a < 42) {
  b = bar(a)
} else {
  b = baz(a)
}
return b
```



Листинг 11 — Примеры промежуточного представления, синим цветом отмечены закрепленные инструкции, зеленым – плавающие

3 Реализация функционального расширения промежуточного представления

Как было показано Ричардом Келси [27], программу, написанную с использованием SSA, всегда можно переписать в эквивалентную, использующую стиль передачи продолжений. Это преобразование является *синтаксическим*, то есть записывает те же семантические действия, но с использованием других синтаксических конструкций.

Базовый блок b промежуточного представления сопоставляется продолжению f, φ-функции b соответствуют параметрам f, безусловные переходы в блок b соответствуют вызовам продолжения f, аргументы φ-функций становятся аргументами соответствующих вызовов.

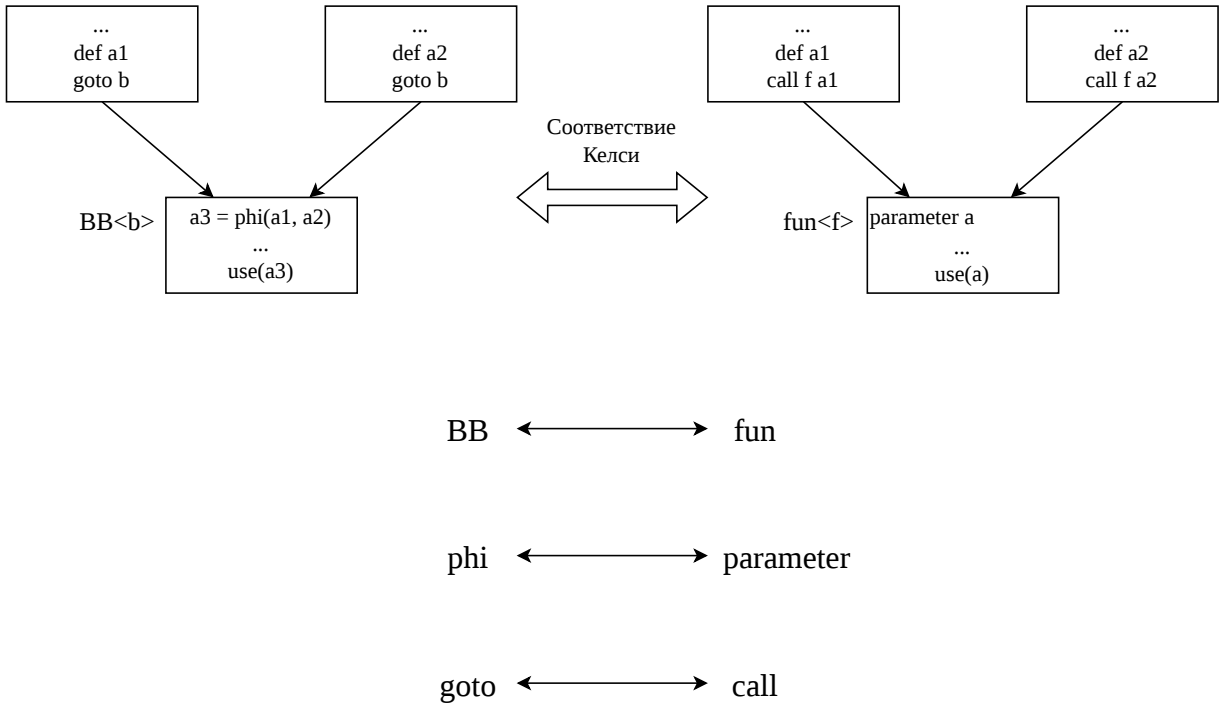


Рисунок 4 — Соответствие Келси

В статье «A correspondence between continuation passing style and static single assignment form» [27], также утверждается, что подобное

преобразование может быть выполнено без проведения глобальных анализов.

Обратное преобразование из CPS в SSA не требует проведения глобальных анализов только если трансформируемая часть программы не содержит продолжений высших порядков[2].

Для поддержки продолжений, согласно соответствию Келси, были добавлены следующие инструкции:

1. `Param` – инструкция, находящаяся в начале линейного участка. является параметром блока. Это новая точка схождения данных. Как и φ-функции параметры всегда располагаются в начале блока.
2. `TailCall` – инструкция хвостового вызова, завершающая линейные участки. Инструкции необходимо указать в какой блок передается управление и передать аргументы вызова.

Но этого недостаточно для поддержки продолжений высшего порядка. Для их реализации была добавлена возможность передавать блоки в хвостовые вызовы по значению. То есть блоки промежуточного представления теперь являются сущностями первого порядка.

В ходе исследования было замечено что трансформация может производиться локально и не требует конвертации всего промежуточного представления в CPS. Это дает возможность использовать CPS только там где это необходимо. Локальное преобразование можно увидеть на рисунке 5.

[2]Продолжение называется продолжением высшего порядка, если хотя бы один его аргумент является продолжением

Определение 3.1 (блок-функция). Базовый блок промежуточного представления является блоком-функцией, если он не использует φ-функции.[3]

Несмотря на то что преобразование порождает эквивалентное промежуточное представление, в некоторых случаях оно может быть более подходящим чем SSA-форма. В ходе трансформации разбиваются дуги потока данных в точках схождения управления, явная передача управления по дугам графа заменяется на неявные переходы с помощи `tailCall`. Это облегчает реализацию структурных преобразований промежуточного представления, например вынос участков промежуточного представления в независимые функции.

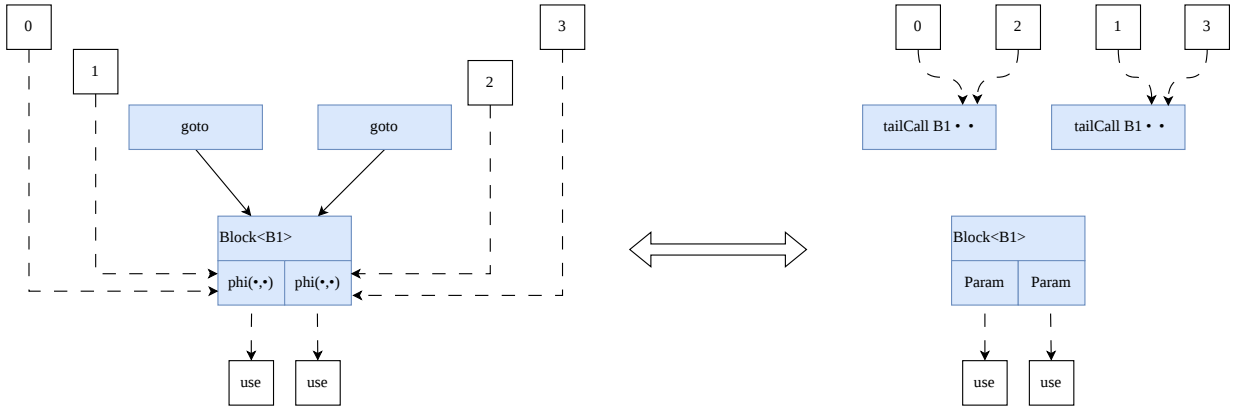


Рисунок 5 — Блок промежуточного представления до трансформации и после

Интеграция CPS-блоков в конвейер оптимизирующей компиляции потребовала дополнительных усилий – появилась необходимость восстанавливать поток управления для неявных переходов через

[3]Блоки, не использующие ни φ-функции, ни параметры блоков могут быть отнесены как к обычным блокам, так и к блокам-функциями, взависимости от контекста.

хвостовые вызововы, также понадобилось поддержать корректный анализ потока данных, проходящего через инструкции `Param` и `TailCall`.

Для обеспечения консистентности промежуточного представления при работе с CPS-расширением требуется дополнить текущие инварианты компилятора. Были добавлены следующие инварианты:

1. Блок является либо блоком-функцией, либо использует φ-функции. Блок одновременно использующий и φ-функции, и параметры не является корректным.
2. В блок-функцию управление может быть передано только при помощи инструкций `TailCall`. В блоки, не использующие φ-функции и параметры, разрешается передавать управление как при помощи `Goto`, так и при помощи `TailCall`.

Поддержка функциональных элементов в промежуточном представлении открывает возможности для исследования применимости функциональных оптимизаций и преобразований к императивным программам. Также становится возможной поддержка функциональных замыканий на уровне промежуточного представления, что открывает новые возможности для их оптимизаций [28].

3.1 Регионы–функции промежуточного представления

Преобразования подъема и специализации лямбда-функций (англ. lambda-lifting, lambda-dropping) являются основой для множества оптимизаций функциональных языков программирования. Преобразование подъема лямбда-функции заключается в исключении свободных переменных путем добавления новых параметров, что

позволяет вынести функцию из контекста объемлющей. Обратное преобразование наоборот, специализирует параметры функции определенным контекстом. Пример подъема и специализации функции можно найти на рисунке 6.

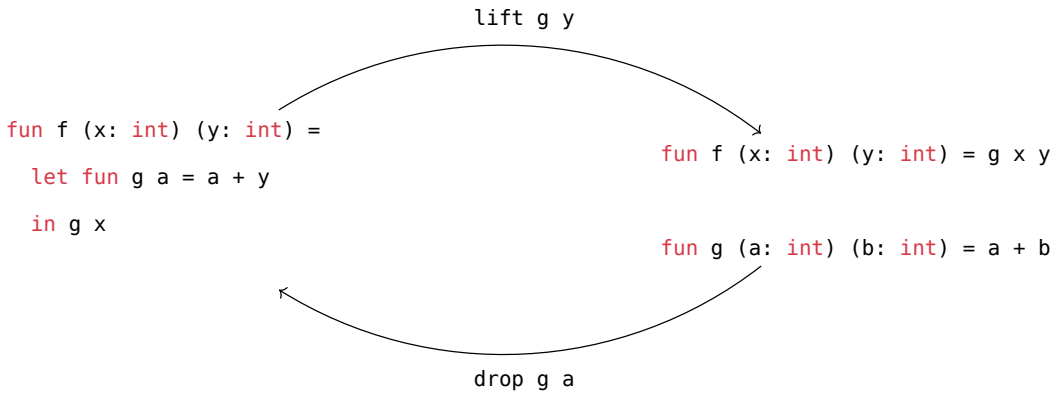


Рисунок 6 — Подъем и специализация лямбда-функции

Через подъем и специализацию выражаются многие другие трансформации промежуточного представления [24].

Однако для выполнения таких преобразований необходимо анализировать функцию целиком, а не только её входной блок. Требуется также учитывать блоки, зависящие от выбранного. В классических функциональных промежуточных представлениях эта задача упрощается тем, что вложенность функций выражается явно. Поэтому возникает вопрос: как определить аналоги функций в смешанном промежуточном представлении?

Попробуем переложить интуитивные свойства функций в языках программирования на промежуточное представление У функции есть единственная точка входа, также функция образует область видимости, то есть переменные, объявленные в функции, не могут быть

использованы за её пределами. Подграф графа, обладающий этими свойствами будем называть регион-функцией.

Определение 3.1.1 (Регион-функция). Регионом-функцией мы будем называть подграф графа потока управления F, обладающий следующими свойствами:

(1) У F имеется единственный входной блок, то есть существует только один блок, такой что в него входят дуги из дополнения F к CFG. Входной блок будем обозначать как $\mathrm{in}(F)$

(2) Все переменные, размещенные в F корректным алгоритмом планирования инструкций, имеют использования только в F, за исключением использований в φ-функциях.

Из (1) следует, что входной блок региона-функции доминирует все блоки региона функции.

3.2 Независимые подграфы

Любую регион-функцию можно выделить в независимый подграф, таким же образом как можно вынести вложенную функцию из объемлющей в функциональных языках программирования. При этом, для выноса регион-функции не требуется перестройка SSA-формы и другие глобальные анализы. Прежде чем доказать это важное свойство регион-функций, введем несколько определений.

Определение 3.2.1 (Свободная переменная). Свободной переменной подграфа G назовем такую инструкцию, которая имеет хотя бы одно использование в G, но не расположена в G

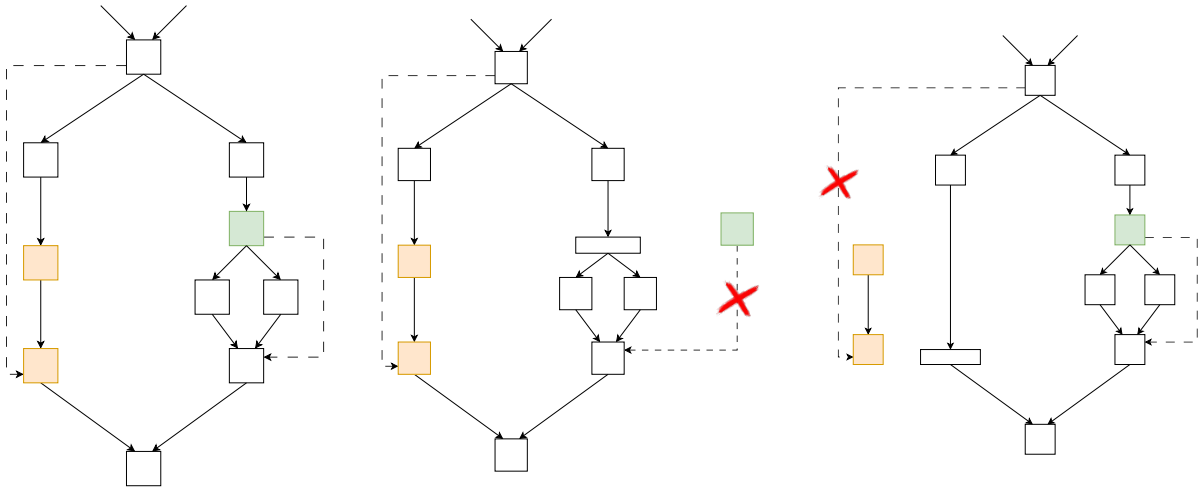
Определение 3.2.1 является более общим аналогом определения свободных переменных для лямбда-функций.

Для того чтобы выделить некоторый регион в независимый подграф необходимо убедиться, что такое преобразование будет корректным.

Для наглядности возьмем участок CFG, изображенный на рисунке 7(а) и рассмотрим на нем случаи, в которых выделение региона CFG в независимую функцию является некорректным. Блок, отмеченный на рисунке 7(б) зеленым цветом, объявляет переменную, имеющую использования за пределами этого блока. Поэтому выделение этого блока в независимую функцию нарушит корректность промежуточного представления. Заметим, что подобные ситуации невозможны регион-функции.

Множество блоков, отмеченное оранжевым цветом на рисунке 7(с), является регион-функцией. Но последний блок оранжевого подграфа содержит свободную переменную, поэтому его нельзя вынести в независимую функцию без потери корректности промежуточного представления.

То есть подграф, который может быть выделен в независимую функцию должен быть полностью изолирован от внешнего контекста. Такие подграфы будем называть *независимыми*.



(а) (б) (с)

Рисунок 7 — Участок CFG, штрихованными стрелками обозначены использования инструкций между блоками

Определение 3.2.2 (Независимый подграф). Подграф промежуточного представления G, называется независимым подграфом если

(1) в G не используются переменные, объявленные в дополнении $\overline{G}$. То есть G не содержит свободных переменных.

(2) Все инструкции $\overline{G}$, за исключением φ-фукнций, не используют инструкции, объявленные в G.

Независимые подграфы можно вынести в функции промежуточного представления, не нарушив корректности. Но независимые регионы редко встречаются в промежуточных представления реальных программ, их необходимо формировать.

Регион-функции уже соответствуют свойству 3.2.2(2). Для того чтобы сформировать из регион-функции независимый подграф необходимо исключить свободные переменные.

По аналогии со свободными переменными лямбда-функций, свободные переменные регион-функции можно исключить при помощи параметров блоков-функций.

Алгоритм 3.2.3 (Исключение свободных переменных)

Вход: регион-функция с входным блоком b и список свободных переменных FV.

1. Трансформировать входной блок в блок-функцию
2. Для свободной переменной $v_i \in \text{FV}$ добавить во входной блок параметр p_i
3. Все использования v_i внутри регион-функции заменить на p_i
4. Повторить для всех свободных переменных FV

Такое преобразование может нарушить SSA-свойство программы во всем промежуточном представлении. Это показано на рисунке 8. Но внутри региона SSA-свойство не нарушается.

Свойство 3.2.4. Исключение свободных переменных регион-функции не требует перестройки SSA-формы внутри регион-функции.

Доказательство. Пусть F – регион-функция. F имеет единственный входной блок b. А значит все пути, по которым свободные переменные достигают использований в F проходят через b. Параметры добавляются во входной блок, а значит для любого использования параметра p внутри регион-функции есть только одно достигающее определение и перестройка SSA-формы не требуется. □

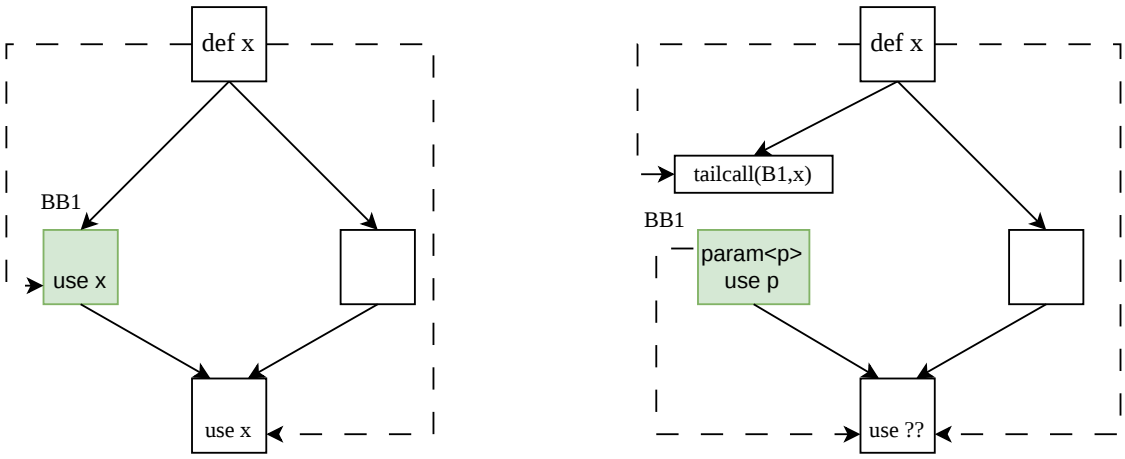


Рисунок 8 — Исключение свободной переменной может привести к нарушению SSA-формы.

Свойство 3.2.4 утверждает, что перестройка SSA-формы внутри регион-функции не требуется, но что делать с нарушением SSA-свойств объемлющего графа? Можно заметить, что после исключения свободных переменных регион-функция стала независимым подграфом и её можно безопасно вынести. Все новые определения, появившиеся во время исключения свободных переменных, будут также вынесены из объемлющего графа. Это значит, что SSA-форма более не будет требовать перестройки.

Пример исключения свободных переменных в регион-функции, нарушающий SSA-свойство промежуточного представления можно увидеть на рисунке 9. Также на этом рисунке изображено как SSA-свойство восстанавливается после выделения регион-функции в независимый подграф.

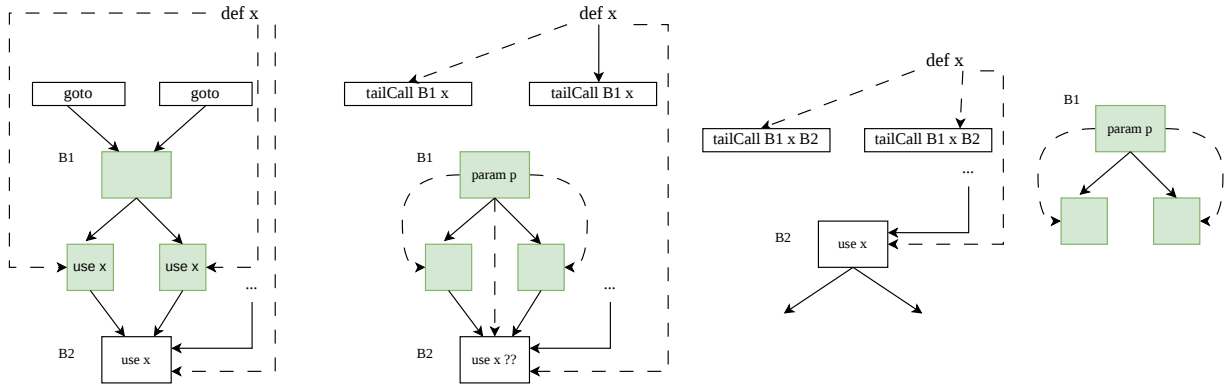


Рисунок 9 — После исключения свободной переменной, в графе осталось использование инструкции `x`, доминируемое последней версией `p`. После вынесения регион-функции, восстановление SSA не требуется.

Для произвольных подграфов с несколькими входами исключение свободных переменных в большинстве случаев требует достройки SSA-формы. Вставка хотя бы одной новой φ-функции может привести к необходимости глобальной перестройки SSA-формы во всем промежуточном представлении.

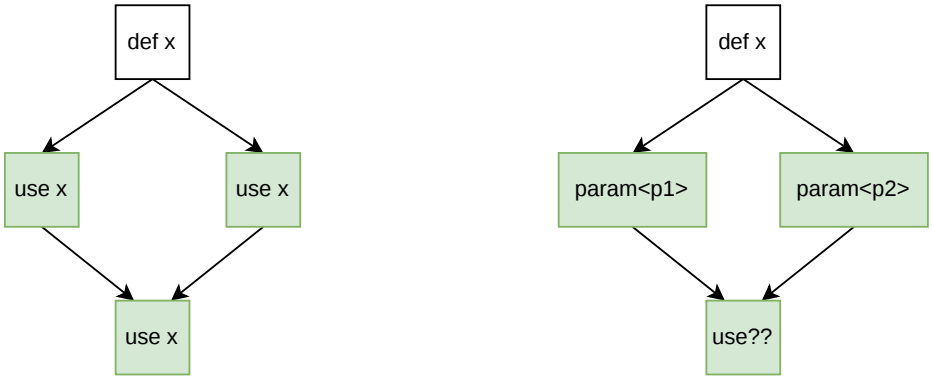


Рисунок 10 — Перестройка SSA-формы для произвольного региона

Таким образом регион-функции являются естественным аналогом функций в расширенном промежуточном представлении, ведь именно для этих множеств можно эффективно производить операции подъема и специализации.

Заметим, что для фиксированного блока b могут существовать несколько различных регион-функций. В следующих разделах мы приведем описание различных множеств, удовлетворяющих свойствам регион-функций, и опишем алгоритмы их нахождения.

3.2.1 Область доминирования

Любая область доминирования удовлетворяет свойству 3.1.1(1).

Определение 3.2.5 (Область доминирования блока).

Пусть b – блок промежуточного представления p. Тогда множество

$$D(b) = \{b' \in p \mid b \text{ dom } b'\}$$

Будем называть областью доминирования блока b

Но область доминирования блока b также удовлетворяет и 3.1.1(2), в силу свойства доминирования SSA-формы. Если инструкция d, размещенная в $D(b)$, имеет использование u, то по свойству SSA-формы $d \text{ dom } u$, $b \text{ dom } d$, а значит $b \text{ dom } u \Rightarrow u \in D(b)$ так как доминирование транзитивно.

Таким образом, область доминирования любого блока всегда является регионом-функцией. Кроме того область доминирования является максимальной регион-функцией для фиксированного входного блока.

Предложение 3.2.6. *Область доминирования является максимальной по включению регион-фукницией среди всех регион-функций с фиксированным входным блоком.*

Доказательство. Рассмотрим $D(b)$, если блок x не находится в $D(b)$, то $\neg(b \text{ dom } x)$, а значит $D(b) \cup \{x\}$ не является регион-функцией так как не удовлетворяет свойству 3.1.1(1). □

Также область доминирования не зависит от выбора алгоритма размещения плавающих инструкций.

Области доминирования, как будет показано далее, удобно использовать в качестве регион-функций для выноса как можно большего количества кода из промежуточного представления.

Но существуют структурные преобразования, для которых, наоборот, важно включать в регион-функцию как можно меньше инструкций.

3.2.2 Лексический регион

В классических функциональных языках программирования вложенные функции могут использовать переменные, объявленные в объемлющей функции. В таких случаях говорят, что функция замыкает контекст объемлющей. Важно отметить, что если функция g замыкает переменные f, то g обязана быть вложена в f.

Блоки промежуточного представления также могут замыкать инструкции из внешнего контекста, а это значит, что вложенность блоков, формирующих тело функции внутри промежуточного представления по аналогии можно восстановить при помощи def–use цепочек инструкций.

Оказывается, что регионы, построенные по def–use цепочкам, являются минимальными по включению регион-фукнциями для

фиксированного входного блока. Вслед за интуицией о лексических областях видимости, пришедшей из функциональных языков программирования будем называть такие регионы лексическими.

Определение 3.2.7 (Лексический регион). Минимальную по включению регион-функцию для фиксированного входного блока b будем называть лексическим регионом.

Лексический регион для входного блока b будем строить конструктивно с использованием алгоритма на листинге 12.

Алгоритм 3.2.8 (Распознавание лексического региона)

Вход: Входной блок лексического региона b.

Результат: лексический регион со входной вершиной b и множество всех инструкций, объявленных в нем.

Сформируем множество блоков, удовлетворяющих 3.1.1(2). Для этого будем обходить def-use цепи инструкций, начиная с инструкций, объявленных в b. Использования в φ-функциях во время обхода не учитываются. Если во время обхода встречается закрепленная инструкция, объявленная в блоке b', то нужно обойти def-use цепи всех инструкций, закрепленных в блоке b'.

Псевдокод алгоритма можно найти на листинге 12. Цикл в строках 9-21 совершает обход def-use связей.

Но после этого обхода лексический регион может остаться несформированным. На рисунке 11(a) показан участок CFG, для

входного блока которого требуется определить лексический регион. На рисунке 11(б) отмечены блоки, размеченные поиском по def-use цепям. Неразмеченными могут остаться блоки, соединяющие размеченные, такие блоки также нужно добавить в лексический регион. Иначе регион будет иметь несколько входных вершин.

Вычислим все выходные блоки региона. Цикл вычисляющий выходные блоки показан на строках 26-30 листинга 12. После обойдем граф по предкам этих блоков, пока не встретим входной блок b. Если по пути будет встречен блок, не добавленный в лексический регион, добавим его. Эта часть алгоритма показана в строках 33-40на листинге 12. Блоки, добавленные таким образом можно увидеть на рисунке 11(c).

```
def lexReg(b: Block): (Set[Block], Set[ValueNode]) = {
  // Worklist - список который можно изменять во время итерации
  val worklist = Worklist(b.spine ++ b.phies ++ b.params)
  val visitedBlocks = mutable.Set.from[Block](b)
  val visitedIns = mutable.Set.empty[Node]

  // Во время итерации инструкции удаляются из списка
  // Цикл продолжается до тех пор пока список не опустеет
  for (n <- worklist.drain if !(visitedIns contains n)) {
    visitedIns += n
    val directUses = n.valueUses.filterNot(_.isInstanceOf[Phi])
    for (u <- directUses) {
      match u {
        case b: FloatingNode => worklist += b
        case c: ControlNode if !visitedBlock(c.block) =>
          visitedBlocks += c.block
          worklist ++= (c.block.spine ++ c.block.params)
        case _ =>
      }
    }
  }

  val closure = Worklist.empty[Block]
  val visitedBackward = Worklist.empty[Block]

  for (b <- visitedBlocks) {
    if (b.succs.exists(x => !(visitedBlock contains x))) {
      exits += b
    }
  }

  for (b <- closure.drain if !visitedBackward(b) && b != b) {
    visitedBackward += b
    if (!visitedBlocks(b)) {
      visitedBlocks += b
      visitedIns ++= (b.spine ++ b.params)
    }
    closure ++= b.preds()
  }

  (visitedBlocks, visitedIns)
}
```

Листинг 12 — Алгоритм распознавания лексического региона блока b

Предложение 3.2.9. *Алгоритм 3.2.8 строит минимальный по включению лексический регион для фиксированного входного блока b.*

Доказательство. Рассмотрим цикл обхода региона по def-use связям. Он завершается за конечное число шагов, так как каждая инструкция просматривается ровно один раз, это гарантирует проверка в строке 9. Обход начинается с инструкций объявленных во входном блоке b. Значит в силу свойств SSA-формы и транзитивности отношения доминирования каждый блок, добавленный во время обхода def-use связей, доминируется входным блоком b.

Цикл в строках 32-39 завершается, так как по доказанному все выходы региона доминируются входным блоком b.

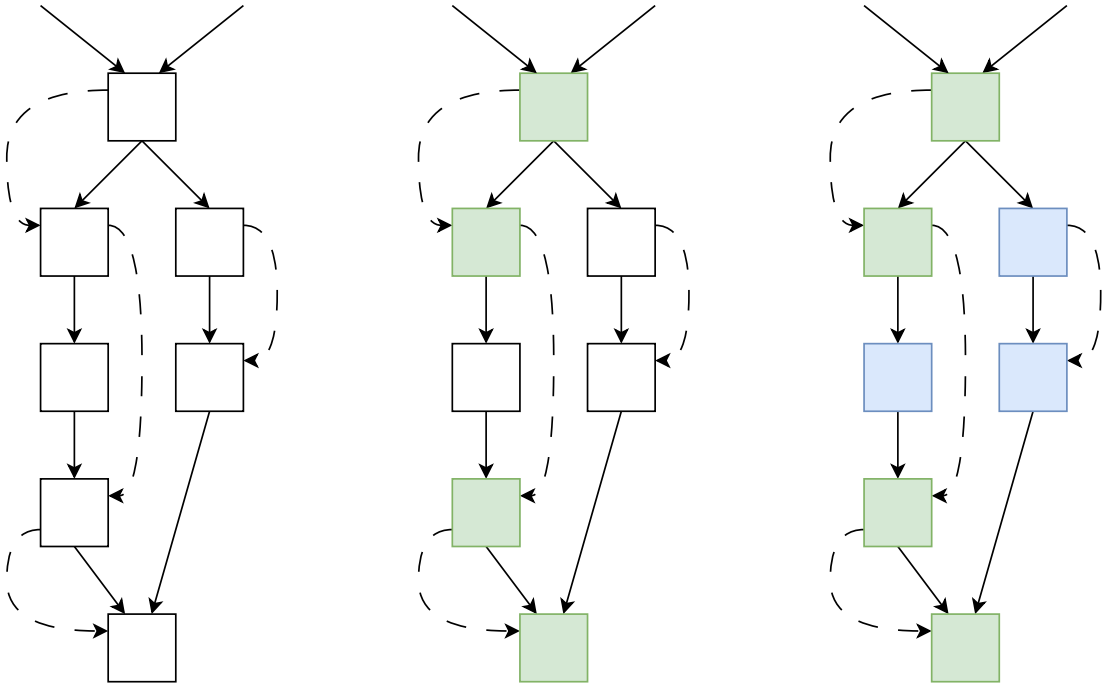
Регион является минимальным по построению, на каждом шаге алгоритма добавляются только те блоки, которые необходимы для выполнения свойств 3.1.1(1) и 3.1.1(2).

Вместе с блоками алгоритм собирает информацию о всех объявленных в регионе инструкциях. Так как поиск по def-use связям начинается с инструкций, объявленных в b, множество найденных инструкций будет содержать только те плавающие инструкции, верхняя грань которых доминируется входным блоком b. □

Зная список инструкций, определенных в регионе-функции можно вычислить множество его свободных переменных. Достаточно проверить аргументы этих инструкций, если они не содержатся в регионе, то являются свободной переменной.

Структурные преобразования промежуточного представления, как правило, требуют копирования инструкций. Поэтому для их реализации удобнее всего использовать именно лексические регионы в силу их

минимальности. Алгоритм 3.2.8 распознавания лексических регионов обнаруживает только те плавающие инструкции, которые необходимо разместить в регионе, что также минимизирует количество копируемого кода.



(а) (б) (с)

Рисунок 11 — Пример работы алгоритма из листинга 12

(а) – неразмеченный регион

(б) – зеленым отмечены результаты поиска по def-use связям

(в) – синим цветом отмечены блоки, найденные вторым проходом алгоритма

3.3 Частичная компиляция методов

Разные участки исходной программы имеют разную относительную частоту исполнения. Участки кода, которые исполняются часто

будем называть *горячими*, эти участки вносят большой вклад в производительность итоговой программы и, как следствие, требуют более высокого уровня оптимизации. Участки с низкой частотой исполнения называются *холодными*, они вносят незначительный вклад в производительность итоговой программы.

Для горячих участков кода компилятор может применять более агрессивные оптимизации: при планировании открытых подстановок могут увеличиваться бюджеты, более активно использоваться оптимизации, копирующие код. При этом количество кода и время компиляции увеличиваются.

Для того чтобы снизить время компиляции можно применять разные наборы оптимизаций к горячим и холодным регионам промежуточного представления. Но в промежуточном представлении одного метода могут находиться как горячие, так и холодные регионы. Это можно увидеть на рисунке 12. Большинство анализов рассматривает все промежуточное представление целиком, то есть оптимизации применяемые к горячему коду также вынуждены просматривать и холодный.

Для того чтобы решить эту проблему можно разбить промежуточное представление на независимые части и применять к ним разные наборы оптимизаций. Такая техника называется частичной компиляцией (англ. Region Based Compilation)[29–32].

В разделе 3.2 обсуждалось, что операция подъема свободных переменных регион-функции переводит её в независимый подграф. То

есть регион-функции можно выносить из холодного кода не перестраивая при этом SSA-форму.

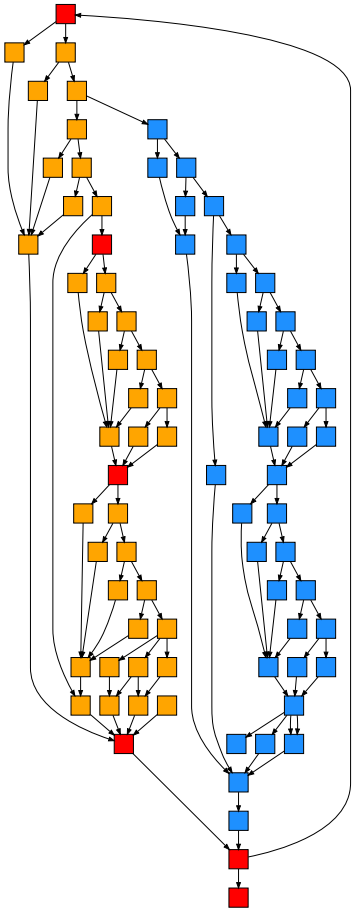


Рисунок 12 — Участок графа потока управления, синим цветом отмечены холодные блоки, оранжевым и красным – горячие.

3.3.1 Разбиение холодного региона на регион-функции

Холодный регион промежуточного представления в общем случае не обязан быть регион-функцией, например он может иметь несколько входных блоков. Но его можно разбить на непересекающиеся регион-функции. При этом логично размещать в получаемых регион-функциях как можно больше инструкций. Во-первых, для того чтобы уменьшить количество инструкций, оставшихся в горячем регионе. Для этого перед формированием регионов функций мы

выполняем планирование инструкций, пытаясь внести в холодный регион как можно больше плавающих инструкций. Во-вторых, чем больше блоков будет содержаться в регионах-функциях, тем меньше их понадобится для покрытия холодного региона и тем меньше будут издержки на их формирование. Поэтому в качестве регион-функций мы будем использовать области доминирования блоков из-за их свойства максимальности.

Для того чтобы сформировать разбиение холодного региона минимальным количеством регион-функций можно использовать алгоритм, изображенный на листинге 3.3.1.

Алгоритм 3.3.1 (Построение минимального разбиения)

Вход: регион холодного кода C

Результат: P – разбиение C минимальным количеством регионов с единственным входом.

Граф потока управления обходится в топологическом порядке T, где b_i – это блок с номером i в топологическом порядке. Если $b_i \in C$, то добавляем $D(b_i)$ в P.

Так как элементы $D(b_i)$ уже добавлены в разбиение их необходимо исключить из просматриваемых блоков T.

После того как все блоки T будут просмотрены алгоритм завершается.

```
def splitColdRegion(coldRegion: Set[Block]): Set[Region] = {
  val topSort = cfg.topSort
  val coldRegions = mutable.Set.empty[Region]

  for (b <- topSort if coldRegion(b)) {
    val dom = D(b)
    coldRegion += dom
    topSort --= dom.blocks
  }

  coldRegions
}
```

Листинг 13 — Псевдокод алгоритма

Предложение 3.3.2. *Алгоритм 3.3.1 строит разбиение холодного региона минимальным количеством областей доминирования.*

Доказательство.

Каждый элемент полученного разбиения является областью доминирования, так как области доминирования либо вложены, либо не пересекаются. Поэтому в строчке 8 удаляются элементы принадлежащие только рассматриваемому $D(b)$.

Алгоритм строит разбиение $P = \{D(b_1), ..., D(b_n)\}$, каждый элемент которого является областью доминирования.

Рассмотрим произвольное разбиение холодного региона P'. Каждый элемент этого разбиения – это регион CFG с единственным входом. Докажем, что любой регион $R \in P'$ содержит не более одного b_i. Если $b_i, b_j \in R$, то $r = \text{in}(R)$ доминирует b_i и b_j. Но тогда r встречается раньше в топологическом обходе, а значит алгоритм с листинга 3.3.1 выбрал бы его в качестве входной вершины для области доминирования, в которой бы содержались b_i и b_j. Получили противоречие, а значит

в каждом R содержится максимум один b_i. Так как P' – разбиение холодного региона оно обязано содержать все b_i, а значит в $|P'| \geq n$.

То есть минимальное количество регионов в разбиении – это количество регионов в разбиении P, построенном приведенным алгоритмом. □

На практике часто случается, что даже среди минимального разбиения встречается слишком много регион-функций, поэтому количество выносимых функций определяется эвристически.

На основе расширенного промежуточного представления был реализован модуль частичной компиляции. Холодный регион разбивается на области доминирования согласно алгоритму 3.3.1. Наиболее крупные области доминирования выносятся в независимые функции, как обсуждалось в 3.2. К каждой функции применяется разный набор оптимизаций.

4 Результаты

4.1 CPS-расширение

Для тестирования корректности расширенного промежуточного представления был реализован специальный режим компиляции. Во время цикла оптимизации выбиралось подмножество блоков промежуточного представления, упорядочивалось, затем к блокам с заранее заданным шагом k применялось соответствие Келси. Такой режим тестирования позволяет получить различные конфигурации промежуточного представления: применение соответствия Келси ко всему промежуточному представлению ($k = 1$, выбираются все блоки), применение соответствия к каждому второму блоку ($k = 2$, все блоки), применение соответствия к блокам, содержащим φ-функции ($k = 1$, выбираются блоки с φ-функциями). После итерации цикла оптимизации промежуточное представление целиком конвертировалось обратно в SSA-форму. Тестирование различных конфигураций промежуточного представления проводилось на полном наборе тестов виртуальной машины Huawei.

Текущая реализация промежуточного представления проходит весь набор тестов при разных конфигурациях тестировочного режима.

Таким образом смешанное промежуточное представление адаптировано к существующим анализам и оптимизациям.

Под адаптацией здесь подразумеваются два разных сценария. Наличие CPS блокирует возможности для оптимизации, но дальнейшая компиляция происходит без ошибок и порождается корректный код. Либо

оптимизация переписывается так, чтобы она могла происходить также как и на обычном представлении.

Большинству оптимизаций требуется информация о потоке управления. В случае использования неявных переходов при помощи инструкций `tailCall`, поток управления нужно достраивать. Но это не всегда можно сделать, например в случае использования продолжений высших порядков. Но даже для продолжений нулевого и первого порядка пришлось изменить большое количество кода, чтобы обеспечить корректность анализов и оптимизаций.

4.2 Модуль частичной компиляции

Модуль частичной компиляции был протестирован на наборе микробенчмарков.

В таблице 1 приведены средние значения для 50ти измерений на тестовом стенде 2, доверительные интервалы с уровнем доверия 95% не пересекаются для времени компиляции со включенным модулем частичной компиляции и с выключенным.

Таблица 1 — Результаты тестирования на микробенчмарках

Бенчмарк	Время, с	Время опт., с	Время отн. %
SimpleDB	48.17	47.2	**−2.06%**
Call-in-cycle	41.05	39.13	**−4.91%**
Lisp	25.34	23.93	**−5.85%**

Бенчмарк `SimpleDB` содержит реализацию простой векторной базы данных с возмощностью поиска наиболее подоходящего вектора и сортировки данных. Бенчмарк `Call-in-cycle` содережит горячий цикл,

вызывающий функцию. Бенчмарк `Lisp` – это интерпретатор простого `lisp`-языка.

Таблица 2 — Характеристики тестового стенда

Процессор	Intel Core i7-780
Память	32GB DDR4

4.3 Выводы

Предложенный подход к построению расширения промежуточного представления с неявными переходами по хвостовым вызовам, оказался слишком трудоемким для поддержки в кодовой базе компилятора.

В будущих версиях этого промежуточного представления для передачи управления планируется использовать дуги потока управления везде, где это возможно. При этом сохранив возможность трансформировать блоки в блок-функции.

Инструменты, разработанные в данной работе, вместе с подходом к оптимзации функциональных замыканий, представленном в работе [28], позволят реализовать различные оптимизаций функциональных замыканий.

Также планируется развивать модуль частичной компиляции и применять его в новых оптимизациях, например для частичной подстановки функций [30].

ЗАКЛЮЧЕНИЕ

Результаты работы:

– Реализовано функциональное расширение промежуточного представления оптимизирующего компилятора
– Режим частичного преобразования промежуточного представления в CPS и его интеграция с оптимизациями и анализами протестированы на представительном наборе тестов.
– Реализованы аналоги операций подъема и специализации лямбда-функций.
– Реализован модуль частичной компиляции с использованием разработанного промежуточного представления и его трансформаций.
– Получено значительное уменьшение времени компиляции на наборе микробенчмарков

Направления дальнейших работ:

– Исследование применимости других существующих функциональных преобразований в предложенном расширении
– Разработка оптимизаций, использующих модуль частичной компиляции

Публикации

1. Кареба Ю. С. Реализация мультипарадигменного промежуточного представления оптимизирующего компилятора. // Материалы 64–й Международной научной студенческой конференции МНСК–2026. Математика. Новосиб. гос. ун-т. Новосибирск, 2026. Доклад отмечен дипломом первой степени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. LLVM Project. LLVM Language Reference Manual. 2025.
2. Gosling J. и др. The Java Language Specification, Java SE 8 Edition. Addison-Wesley, 2015.
3. Evans B. Understanding Java method invocation with invokedynamic // Java Magazine. 2021.
4. Владимиров К. Оптимизирующие компиляторы. Структура и алгоритмы. АСТ, 2024.
5. Rastello F., Tichadou F.B. SSA-based compiler design. Springer, 2022.
6. Click C., Paleczny M. A simple graph-based intermediate representation // ACM Sigplan Notices. ACM New York, NY, USA, 1995. Т. 30, № 3. Сс. 35–49.
7. Duboscq G. и др. Graal IR: An extensible declarative intermediate representation // Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop. 2013. Сс. 1–9.
8. Jones S.L.P., Santos A.M. A transformation-based optimiser for Haskell // Science of computer programming. Elsevier, 1998. Т. 32, № 1–3. Сс. 3–47.
9. Appel A., Jim T. Continuation-passing style, closure-passing style // 16th Ann. ACM Symp. on Principles of Programming Languages. 1989.
10. Adams N. и др. Orbit: An optimizing compiler for Scheme // ACM SIGPLAN Notices. ACM New York, NY, USA, 1986. Т. 21, № 7. Сс. 219–233.

11. Kildall G.A. A unified approach to global program optimization // Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1973. Cc. 194–206.
12. W.Appel A. Modern Compiler Implementation in ML. Cambridge University Press, 1998.
13. Cytron R. и др. Efficiently computing static single assignment form and the control dependence graph // ACM Transactions on Programming Languages and Systems (TOPLAS). ACM New York, NY, USA, 1991. Т. 13, № 4. Cc. 451–490.
14. Braun M. и др. Simple and efficient construction of static single assignment form // International Conference on Compiler Construction. 2013. Cc. 102–122.
15. Click C. Global code motion/global value numbering // Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation. 1995. Cc. 246–257.
16. Johnson N., Mycroft A. Combined code motion and register allocation using the value state dependence graph // International Conference on Compiler Construction. 2003. Cc. 1–16.
17. Pierce B.C. Types and programming languages. MIT press, 2002.
18. Danvy O., Nielsen L.R. A first-order one-pass CPS transformation // Theoretical computer science. Elsevier, 2003. Т. 308, № 1–3. Cc. 239–257.
19. Kennedy A. Compiling with continuations, continued // Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. 2007. Cc. 177–190.

20. Marat A., Mikhail B. Kotlin Language Specification: Asynchronous Programming with Coroutines. 2024.
21. Lindley S. Normalisation by evaluation in the compilation of typed functional programming languages. University of Edinburgh. College of Science, Engineering. School of~…, 2005.
22. Maurer L. и др. Compiling without continuations // Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. Сс. 482–494.
23. Flanagan C. и др. The essence of compiling with continuations // ACM Sigplan Notices. ACM New York, NY, USA, 1993. Т. 28, № 6. Сс. 237–247.
24. Leißa R., Köster M., Hack S. A graph-based higher-order intermediate representation // 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2015. Сс. 202–212.
25. Lattner C. и др. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation // 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2021. № . Сс. 2–14.
26. Tarasov V. HIVM: MLIR Dialect Stack for Ascend NPU Compilation [Электронный ресурс]. 2026. URL: https://llvm.org/devmtg/2026-04/.
27. Kelsey R.A. A correspondence between continuation passing style and static single assignment form // ACM SIGPLAN Notices. ACM New York, NY, USA, 1995. Т. 30, № 3. Сс. 13–22.

28. Болохонов А.В. Специализация функциональных замыканий в ходе оптимизации императивных языков программирования. Новосибирский Государственный Университет, 2024.
29. Hank R.E., Hwu W.-M.W., Rau B.R. Region-based compilation: An introduction and motivation // Proceedings of the 28th Annual International Symposium on Microarchitecture. 1995. Cc. 158–168.
30. Way T., Breech B., Pollock L. Region formation analysis with demand-driven inlining for region-based optimization // Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622). 2000. Cc. 24–33.
31. Suganuma T., Yasue T., Nakatani T. A region-based compilation technique for a Java just-in-time compiler // Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation. 2003. Cc. 312–323.
32. Suganuma T., Yasue T., Nakatani T. A region-based compilation technique for dynamic compilers // ACM Transactions on Programming Languages and Systems (TOPLAS). ACM New York, NY, USA, 2006. Т. 28, № 1. Cc. 134–174.