Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА ИУ-9 «Теоретическая информатика и компьютерные технологии»

ОТЧЁТ ПО ПРЕДДИПЛОМНОЙ ПРАКТИКЕ

НА ТЕМУ:

«Оптимизирующий компилятор диалекта Бейсика для LLVM»

Студент группы ИУ9И-82Б _____ _____
(Подпись, дата) (И.О.Фамилия)

Руководитель _____ _____
(Подпись, дата) (И.О.Фамилия)

2025 г.

О Г Л А В Л Е Н И Е

В В Е Д Е Н И Е

В мире программирования и разработки программного обеспечения компиляторы играют ключевую роль, обеспечивая перевод высокоуровневого исходного кода программы в машинный код, который может быть исполнен целевой вычислительной машиной. Исторически компиляторы были созданы для различных языков программирования, и каждый компилятор обычно имеет свои особенности, оптимизации и специфические подходы к генерации кода.

Использование компиляторов позволяет программистам использовать удобные и понятные для них языки, такие как C, C++, Java, и далее преобразовывать их в машинный код, который может быть исполнен на различных платформах. Это обеспечивает создание сложных программ и систем, которые могут быть эффективно выполнены компьютером.

Кроме того, компиляторы способствуют переносимости программного обеспечения. Они позволяют программистам написать код всего один раз и скомпилировать его для различных платформ, включая Windows, Linux, macOS и другие. Это значительно упрощает процесс разработки, поскольку разработчики могут сосредоточиться на написании кода, не беспокоясь о тонкостях, специфичных для каждой платформы.

Однако данная программисту возможность абстрагироваться от написания низкоуровневых инструкций создает множество трудностей при трансляции программы с исходного языка на целевой язык. Поэтому одним из ключевых аспектов значения компилятора является его способность генерировать оптимизированный машинный код.

Оптимизированный код может значительно улучшить производительность программы, позволяя ей работать быстрее и эффективнее в смысле использования ресурсов компьютера, таких как процессорное время и память. Оптимальный код может использовать доступные ресурсы более эффективно, что ведет к улучшению производительности и уменьшению нагрузки на систему.

1. Анализ поставленной задачи

Необходимо разработать оптимизирующий компилятор для диалекта языка BASIC. Основной целью данного компилятора является генерация оптимизрованного кода целевого языка LLVM IR. Оптимизация генерируемого кода имеет важное значение для разработчиков программного обеспечения, поскольку обеспечивает создание более сложных и эффективных программ и систем, что является ключевым для повышения отзывчивости и производительности приложений. Разработанный компилятор должен решать следующие задачи:

1. Трансляция исходного кода на диалекте языка BASIC в код для целевой платформы, учитывая специфику языка
2. Обеспечивать эффективное преобразование исходного кода, повышая производительность и качество генерируемых программ
3. Сохранение программной совместимости со стандартной библиотекой поддержки времени исполнения для компановки исполняемого файла

2. Схема программного обеспечения

Программная реализация компилятора использует модульную архитектуру с четким разделением на следующие компоненты: препроцессор, парсер, семантического анализатор, генератор промежуточного представления, конвейер оптимизации и генератор целевого кода

На рисунке №1 представлена схема программной реализации компилятора, иллюстрирующая основные компоненты и их всаимосвязи.

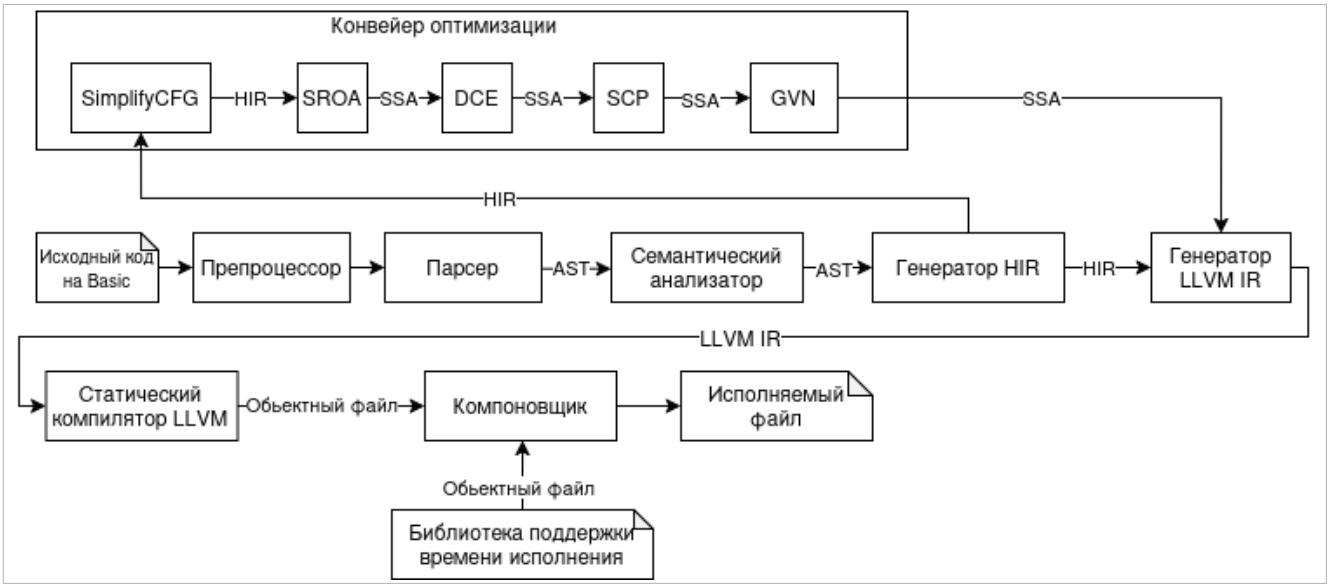

Рисунок 1 - схема ПО

Препроцессор играет важную роль в подготовке исходного кода к последующим этапам компиляции. Его основная задача заключается в обработке макросов и директив включения. В своей реализации данный модуль полагается на модуль парсера, используя отличную от исходного языка грамматику. Результатом работы данного модуля является обработанный исходный код на диалекте языка BASIC. Препроцессор позволяет пользователям компилятора осуществлять модульную разработку и повторно использовать разработанные ранее модули.

Следующим компонентом является парсер. Его основной задача заключается в синтаксическом анализе предобработанного препроцессором исходного кода на диалекте языка BASIC и его преобразовании в абстрактное

синтаксическое дерево, используя грамматику языка. Данный модуль в своей реализаций использует сторонюю библиотеку parser_edsl.

Семантический анализатор является неотъемлемым компонентом в структуре компилятора, обеспечивает проверку корректности программы на семантическом уровне, учитывая специфику языка, и производит первоначальную оптимизацию исходного текста. Входными данными для данного компонента является абстрактное синтаксическое дерево, полученное в результате работы парсера, и результатом работы является оптимизированное абстрактное синтаксическое дерево, наполненное метаинформацией о типах переменных и результирующих типах выражении.

Генератор промежуточного представления диалекта языка BASIC преобразовывает, полученное, в следствии семантического анализа, абстрактное синтаксическое дерево в трёхадресный код промежуточного представления BasicHIR.

Конвейер оптимизации получает сгенерированный трёхадресный код промежуточного представления и производит его трансформации с целью оптимизаций исходного кода программы. Конвейер оптимизаций в свою очередь представляет из себя набор взаимосвязанных между собой подмодулей, производящих оптимизрующие трансформаций промежуточного представления

Следующим компонентом является генератор целевого языка LLVM-IR, который в свою очередь обеспечивает корректное преобразование трёхадресного кода промежуточного представления полученного от генератора промежуточного представления или от конвейера оптимизации в трёхадресный код LLVM-IR.

Неотъемлимой частью диалекта языка BASIC является стандартная библиотека времени исполнения. Основная цель данной библиотеки заключается в предоставлений интерфейса и реализаций методов для работы со строками и операций вывода в выходной поток простых типов диалекта языка BASIC. Стандартная библиотеки времени исполнения реализована с помощью языка программирования C++ и предкомпилирована в обьектный файл

Заключительный этап компиляций программы включает в себя совместную работу двух сторонних компонентов – статического компилятора LLVM и компоновщика. Статический компилятор LLVM преобразовывает трёхадресный код промежуточного представления LLVM-IR в объектный файл для целевой платформы. Компоновщик связывает полученный от статического компилятора объектный файл с объектным файлом стандартной библиотеки поддержки времени исполнения и генерирует итоговый исполняемый файл.

3. Оптимизации абстрактного синтаксического дерева

В рамках разработки данного проекта на этапе семантического анализа были реализованы две локальные оптимизации абстрактного синтаксического дерева – свёртка констант и устранение тривиальных алгебраических операций. Данные оптимизаций имеют место во время семантического анализа бинарных алгебраических операций - **+**, **-**, *****, **/,** бинарных строковых операции - +, и вызова встроенной функций Len

3.1 Свёртка констант

Свёртка констант является одной из самых распространённых и простых форм оптимизации, которая вычисляет значения константных выражении во время компиляции, а не во время выполнения программы. Случаи в которых будет произведена свёртка констант представлены на листинге №1

```
Sub F(a%)
      Dim arr4%(2+2, 3 * (10 + 12 + (1 > 2)))
      Print Len%(arr4%)
      Print "abc" + "def", Len%("Hello world")
End Sub
```

Листинг 1 – Пример в которых будет произведена свёртка констант

Случаи в которых семантический анализатор будет производить свёртку констант распадается на 2 класса – бинарные алгебраические и строковые операций, вызов встроенной функций Len. Во время анализа бинарных операций, определяет типы аргументов и в случае их константности сворачивает бинарные операций на результирующие константы. Во время анализа вызовов функций Len и анализа её аргумента, если семантический анализатор выведет константность размера её аргумента свёрнет вызов функций в константное значение.

3.2 Устранение тривиальных алгебраических операции

Устранение тривиальных алгебраических операции также является важной оптимизацией при преобразовании абстрактного синтаксического дерева, в котором бинарные алгебраические выражения, заменяются на подвыражения.

Семантический анализатор во время анализа бинарных алгебраических операций проверяет принадлежность операций к классу тривиальных алгебраических операций, приведенных на рисунке 2, и заменяет на не константное подвыражение

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

Рисунок 2 - Класс тривиальных алгебраических операций

4. Оптимизации промежуточного представления

Оптимизация промежуточного представления является центральной и главной частью разработки данного проекта, следует за первоначальной оптимизацией абстрактного синтаксического дерева и генерацией промежуточного предствления. В рамках разработки данного проекта были реализованы следующие алгоритмы оптимизации - оптимизация графа потока управления(SimplifyCFG), скалярное замещение(SROA), удаление “мёртвого” кода (DCE), распространение констант (SCP), глобальная нумерация значений(GVN)

4.1 Описание промежуточного представления BasicHIR

Промежуточное представления диалекта языка BASIC(BasicHIR) является линеаризированным абстрактным синтаксическим деревом и представляет из себя статически типизированный трёхадресный код. При проектировании промежуточного представления BasicHIR за основу был взят трёхадресный код представленный в LLVM-IR. На листинге 2 представлена преобразованная в BasicHIR программа

```
@a = global i32 10
@b = global i32 20
@arr4 = global [1 x i32] [i32 0]
define void @Main(ptr %argsP){
entry:
  %argsP.addr = alloca ptr, i32 1
  store ptr %argsP, ptr %argsP.addr
  %arr4.idx = getelementptr [1 x i32], ptr @arr4, i32 0, i32 0
  %arr4.val = load i32, ptr %arr4.idx
  call void @PrintI(i32 %arr4.val)
  ret void
}

define void @__ctor.arr4(){
entry:
  %arrayinit.element.0 = getelementptr i32, ptr @arr4, i32 0
  %a = load i32, ptr @a
  %b = load i32, ptr @b
  %add = add i32 %a, %b
  store i32 %add, ptr %arrayinit.element.0
  ret void
}
```

Листинг 2 – Пример промежуточного представления BasicHIR

4.1.1 Типы данных промежуточного представления BasicHIR

В BasicHIR поддерживаются следующие примитивные типы данных:

1. Целые числа произвольной разрядности

```
i1 ; булево значение
i32 ; 32-разрядное число
i64 ; 64-разрядное число
```

2. Числа с плавающей точкой

```
float, double
```

3. Указатели

```
ptr — указатель на произвольный тип данных
```

Также поддерживаются следующие производные типы данных:

1. Массивы

```
[ число элементов x тип]
[5 x i32 ] — массив из 5и 32-разрядных целых чисел
[16 x double] — массив из 16 длинных чисел с плавающей точкой
```

2. Структуры

```
{Тип, Тип, …}
{i32, double, i32*}
```

3. Функций

```
i32 (i32, i32)
double ({i32, i64, float*}, i32*)
```

Система типов рекурсивна, поэтому можно использовать многомерные массивы, массивы структур, указатели на структуры и функции, и т. д.

4.1.2 Операций промежуточного представления BasicHIR

В BasicHIR были реализованы операций, которые позволяют описать различные вычисления и управляющие структуры.

BasicHIR поддерживает следующий набор арифметических операций:

1. **neg, fneg** - операция отрицания целых чисел и чисел с плавающей точкой

```
%res = fneg float %val
%res.1 = neg i32 %val.1
```

2. **add, fadd** - операция сложения целых чисел и чисел с плавающей точкой

```
%res = fadd float 4.0, %val
%res.1 = add i32 4, %val.1
```

3. **sub, fsub** - операция вычитания целых чисел и чисел с плавающей точкой

```
%res = fsub float %val, 4.0
%res.1 = sub i32 %val.1, 4
```

4. **sdiv, fdiv -** операция деления целых чисел и чисел с плавающей точкой

```
%res = fdiv float %val, 4.0
%res.1 = sdiv i32 %val.1, 4
```

5. **and, or, xor -** побитовые операций «И», «ИЛИ», «Исключающее ИЛИ»

```
%res = and i32 4, %val
%res.1 = or i32 %val.1, 4
%res.2 = xor i32 %val.2, 4
```

Для работы с памятью и указателями в BasicHIR были предусмотрены следующие инструкций:

1. **alloca** - операция выделения памяти определенного типа

```
%ptr = alloca i32
%ptr.1 = alloca i32, i32 10
```

2. **load -** чтение из памяти определенного типа по адресу

```
%val = load i32, ptr %ptr
```

3. **store** - запись в память определенного типа значения

```
store i32 15, ptr %ptr
```

4. **copy** - запись в память определенного типа значения

```
store i32 15, ptr %ptr
```

5. **getelementptr** - операция вычисления адреса структур любой вложенности

```
%val_ptr = getelementptr ptr, ptr %aptr, i64 0, i32 1
```

Также в BasicHIR были предусмотрены операций приведения типов:

1. **sext, fpext -** приведения целого числа и числа с плавающей точкой к типу большей размерности

```
%X = zext i32 257 to i64
%Y = sext i8  -1 to i16
%Z = fpext float 3.125 to double
```

2. **sitofp** - приведения целого числа без знака, со знаком и числа с плавающей точкой друг к другу

```
%W = sitofp i32 257 to double
```

Для сравнения в BasicHIR используется операций - **fcmp, icmp**

```
%res = icmp eq i32 4, 5
%res.1 = fcmp oeq float 4.0, 5.0
```

Для управления потоком выполнения программы используются следующие инструкций:

1. **call** - вызов функций

   ```
   %res = call i32 @func(i32 %val)
   call void @func.1(i8 97)
   ```
2. **ret** - операция возвращения потока управления из функций к вызывающему

   ```
   ret i32 5
   ret void
   ```
3. **br** - операция переключения потока управления к другому базовому блоку

   ```
   br i1 %cond, label %IfTrue, label %IfFalse
   ```
4. **phi** - операция схождения потока управления

   ```
   %0 = phi i32 [1, %entry], [%6, %while.body]
   ```

4.2 Оптимизация графа потока управления

Оптимизация графа потока управления является одним из ключевых этапов в процессе компиляции, направленных на улучшение производительности и эффективности сгенерированного машинного кода. Граф потока управления представляет собой абстрактное представление программы, отображающее все возможные пути выполнения через различные блоки кода.

В рамках разработки данного проекта были реализованы следующие оптимизации графа потока управления - удаление недостижимых базовых блоков, объединение блоков кода, упрощение ветвлений. Приведённые оптимизаций применяются до тех пор пока изменятся граф потока управления, пошагово его упрощая.

На этапе удаление недостижимых базовых блоков, обращаясь к графу потока управления, оптимизация удаляет те базовые блоки, которые не достижимы из входного базового блока и для которых не существует предшественника

Следующим этапом является обьединение базовых блоков. На данном этапе будет произведён анализ каждого базового блока и его преемника или приемников, в случае когда осуществляются безусловные переходы или условные переходы в базовые блоки не содержащие обращения к памяти, производится их объединение и последующая трансформация инструкций передачи управления, если это необходимо

Заключительным этапом данной оптимизаций является упрощение ветвлении. Различаются 2 случая для которых будет производится оптимизация. В случае безусловного перехода базового блока к своему преемнику, если преемник представляет себя пустой базовый блок с безусловным переходом, то происходит замена преемника на преемника преемника. В случае условного перехода осуществляется оптимизация при константности условного перехода базового блока, при идентичных преемниках, при наличий пустого преемника с безусловным переходом, при наличий пустого преемника с условным переходом

На рисунке №3 представлен один из случаев для которого будет произведено упрощение ветвлении

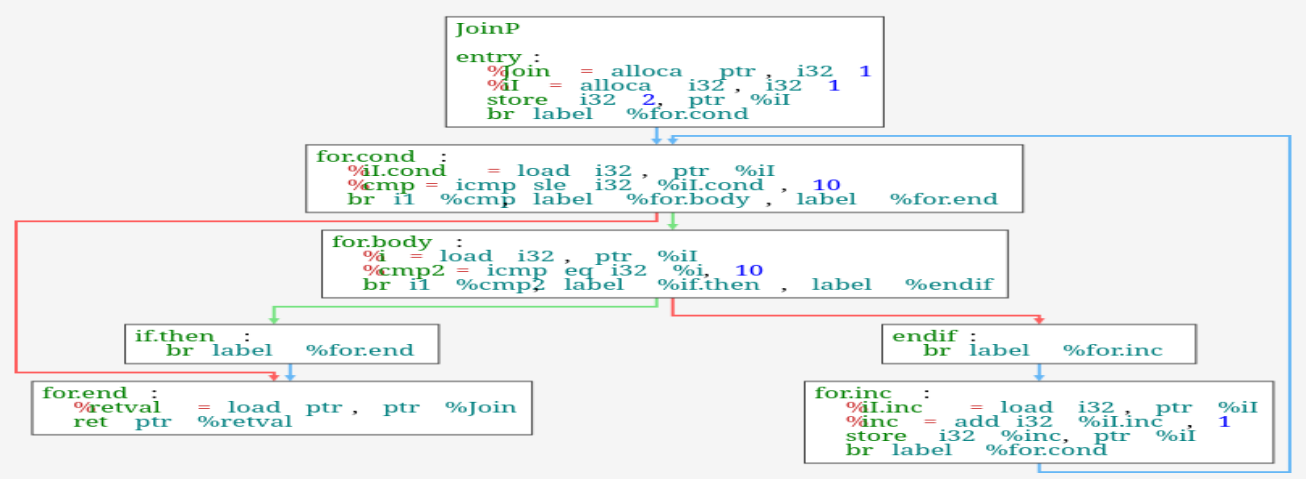

Рисунок 3 - Пример BasicHIR до упрощения ветвлении

4.3 Скалярное замещение

Оптимизация скалярного замещения, разработанная в ходе работы над проектом, фокусируется на замене указателей на индивидуальные скалярные переменные там, где это возможно. Данная оптимизация способствует

уменьшению количества обращений к памяти, улучшению использования регистров процессора и облегчает применение других оптимизаций.

Все доступы к локальным переменным, используемым внутри BasicHIR, делаются через память, то есть через записи в указатели. Данная оптимизация с целью максимального подъема семантической сети распутывает обращения к указателям и заменяет где это возможно на скалярные значения.

В процессе данной оптимизаций происходит трансформация промежуточного представления к его SSA форме, в которой каждая переменная присваивается только один раз.

Основная концепция алгоритма построения SSA формы заключается в расмотренний каждой переменной по отдельности, в дальнейшем для каждой переменной находим все базовые блоки, в которых она видима и определена. Для каждого найденного базового блока вставляется phi узел для рассматриваемой переменной в каждый базовый блока из его фронта доминирования. На листинге №3 представлен алгоритм построения фронта доминирования для функций

```
@staticmethod
def dom_front(function: hir_values.Function):
    dom = DomTree.dom(function)
    idom = {}
    for block, nodes in dom.items():
        for node in nodes:
            idom[node] = block
    dom_front = defaultdict(set)
    cfg_nodes = CFGNode.build_nodes(function)
    for block in function.blocks:
        for pred in cfg_nodes[block].parents:
            current = pred
            while current != idom[block]:
                dom_front[current].add(block)
                if current == function.blocks[0]:
                    break
                current = idom[current]
    return dom_front
```

Листинг 3 – Реализация алгоритма нахождения фронта доминирования CFG

Далее просходит переименование или версионирование переменных, с целью сохранения свойства доминирования. Переименование переменных происходит в порядке обхода в глубину дерева доминаторов.

Заключительным этапом данной оптимизаций является удаление более не используемых выделении памяти. На листинге №4 и №5 представлены примеры BasicHIR до и после скалярного замещения

```
define ptr @JoinP(){
entry:
  %Join = alloca ptr, i32 1
  %iI = alloca i32, i32 1
  store i32 2, ptr %iI
  br label %for.cond
for.cond:
  %iI.cond = load i32, ptr %iI
  %cmp = icmp sle i32 %iI.cond, 10
  br i1 %cmp, label %for.body, label %for.end
for.inc:
  %iI.inc = load i32, ptr %iI
  %inc = add i32 %iI.inc, 1
  store i32 %inc, ptr %iI
  br label %for.cond
for.body:
  %i = load i32, ptr %iI
  %cmp2 = icmp eq i32 %i, 10
  br i1 %cmp2, label %for.end, label %for.inc
for.end:
  %retval = load ptr, ptr %Join
  ret ptr %retval
}
```

Листинг 4 – Пример BasicHIR до построения SSA

```
define ptr @JoinP(){
entry:
  br label %for.cond

for.cond:
  %0 = phi i32 [2, %entry], [%5, %endif]
  %2 = icmp sle i32 %0, 10
  %3 = icmp eq i32 %0, 10
  %4 = select i1 %2, 0, %3
  br i1 %4, label %for.end, label %endif

endif:
  %5 = add i32 %0, 1
  br label %for.cond

for.end:
  ret ptr null
}
```

Листинг 5 – Пример BasicHIR после построения SSA

4.4 Удаление “мёртвого” кода

Удаление мёртвого кода (Dead Code Elimination, DCE) - это одна из фундаментальных техник оптимизации в компиляторах, направленная на выявление и удаление частей программы, которые не влияют на её конечный результат. Применение DCE способствует уменьшению размера исполняемого

кода, сокращению времени компиляции и повышению общей производительности программы.

Алгоритм удаления мёртвого кода работает с результатми предыдущих оптимизаций, используя SSA форму полученную после скалярного замещения.

Изначально для каждого SSA значения устанавливается счётчик, содержащий информацию о том, сколько других значений его используют. Далее для всех значений, в которых этот счётчик равен нулю, определение этого значения удаляется, и для всех значений которые оно использовало, счётчик уменьшается. На листинге №6 представлен алгоритм реализованный в ходе разработки проекта

```
@staticmethod
def transform(function: hir_values.Function):
    ssa_cnt = {}
    for ssa_node in SSA.ssa_nodes(function):
        ssa_cnt[ssa_node] = 0
    for ssa_value, cnt in ssa_cnt.items():
        for block in function.blocks:
            for instr in block.instructions:
                if instr != ssa_value:
                    ssa_cnt[ssa_value] += 1 if ssa_value in instr.operands else 0
    while DCEFunctionTransform.__ssa_cnt_with_value(ssa_cnt, 0) > 0:
        for instr, cnt in ssa_cnt.items():
            if cnt == 0:
                for operand in instr.operands:
                    if operand in ssa_cnt:
                        ssa_cnt[operand] -= 1
                instr.parent.instructions.remove(instr)
                ssa_cnt.pop(instr)
```

Листинг 6 - Алгоритм DCE: удаление мёртвого кода

4.5 Распространение констант

Оптимизация распространения констант (Constant Propagation, SCP) направлена на выявление и замену переменных, которым присваиваются постоянные значения, непосредственно этими постоянными значениями в дальнейшем коде программы. Это позволяет уменьшить количество операций и улучшить производительность итогового машинного кода.

Распространение констант заключается в анализе программы с целью определения переменных, значения которых могут быть установлены на этапе компиляции, и последующей замене этих переменных на их известные константы.

Такая замена снижает количество загрузок и сохранений переменных в память, оптимизирует использование регистров процессора и упрощает дальнейшие этапы оптимизации.

Алгоритм распространения констант работает с результатми предыдущих оптимизаций, используя SSA форму полученную после скалярного замещения и граф зависимости SSA значений

Его основной идеей является продвижение констант сразу по рёбрам зависимостей по данным. Каждый лист графа зависимостей анализируется на предмет того является ли он константой и тогда его значение в решётке - значение константы, в ином случае он **overdefined** (то есть принимает значение $\top$). Всем узлам графа зависимостей не являющихся листьями изначально присваивается значение **undefined** (то есть $\bot$). Далее строится список ребёр в которых один из его концов не принимает значение **overdefined**. Решёточная операция сбора(meet function) выполняется на ребре над значениями его концов и записывается в значение точки использования. Если она привела к изменению в этой точке, то обновляются все инструкций, использующие результат изменившейся инструкций. На листинге №7 представлен оператор/операция сбора используемый в разработанном алгоритме

```
@staticmethod
def __meet(lhs, rhs):
    if lhs == SCPFunctionTransform.SCPLatticeValue.BOTTOM or rhs ==
SCPFunctionTransform.SCPLatticeValue.BOTTOM:
        return SCPFunctionTransform.SCPLatticeValue.BOTTOM
    if lhs == SCPFunctionTransform.SCPLatticeValue.TOP:
        return rhs
    if rhs == SCPFunctionTransform.SCPLatticeValue.TOP:
        return lhs
    if lhs == rhs:
        return lhs
    return SCPFunctionTransform.SCPLatticeValue.BOTTOM
```

Листинг 7 – Оператор сбора значений решётки

4.6 Глобальная нумерация значений

Оптимизация глобальной нумерации значений (Global Value Numbering, GVN) направленна на выявление и устранение избыточных вычислений в программе. Цель GVN — определить, какие выражения всегда вычисляются с

одинаковыми значениями, и заменить их на одну вычисляемую переменную, тем самым сокращая количество операций и улучшая производительность генерируемого кода.

Алгоритм глобальной оптимизации нумерации значений начинается с предположения о том, что все инструкции с одним и тем же оператором имеют тот же выходной результат. Все инструкций разбиваются на пакеты, изначально состоящие из инструкций с общим оператором. Для каждого неизвестного SSA-значения формируется отдельный пакет, состоящий только из него. Две инструкций считаются одинаковыми, если они принадлежат одному пакету и все их операнды попарно принадлежат одинаковым пакетам. Дополнительно phi-узлы считаются одинаковыми, если все операнды принадлежат одному пакету и phi-узлы принадлежат одному базовому блоку. Если мы обнаружили что две инструкции лежащие в одном пакете не эквивалентны, пакет надо разбить по классам эквивалентности. Если в конце разбиения две операции эквивалентны, и одна доминирует над другой, то второе значение избыточно и может быть заменено на первое. На листинге №8 представлено разработанное в ходе работы над проектом разбиение пакетов по классам эквивалентноти

```
def __split(self, function: hir_values.Function):
    changed = True
    while changed:
        changed = False
        new_partition = defaultdict(set)
        for key, value in self.partition.items():
            packet = value.copy()
            instr = packet.pop()
            packetI, packetNonI = set(), set()
            packetI.add(instr)
            while len(packet) > 0:
                j = packet.pop()
                if self.__match(instr, j):
                    packetI.add(j)
                else:
                    packetNonI.add(j)
            new_partition[cnt++] = packetI
            if len(packetNonI) > 0:
                new_partition[cnt++] = packetNonI
                changed = True
        self.partition = new_partition
```

Листинг 8 – Оператор сбора значений решётки

5.Тестирование

5.1 Свёртка констант

Тестирование свёртки констант произовдилось на примере программы представленного на листинге №1. Результат выполнения данной оптимизации, представленный на листинге №9, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
SubroutineDef <line:1, col:1> 'F' 'void(Integer)'
|-Variable <line:1, col:7> 'a' 'Integer'
`-CompoundStatement <line:2, col:5>
  |-VariableDecl <line:2, col:5> 'arr4' 'Integer[4,66]'
  |-VariableDecl <line:3, col:5> 'c' 'Integer[3]'
  | `-InitializerList <line:3, col:17> 'Integer[3]'
  | |-ConstExpr <line:3, col:18> 'Integer' '1'
  | |-ConstExpr <line:3, col:21> 'Integer' '2'
  | `-ConstExpr <line:3, col:24> 'Integer' '3'
  |-VariableDecl <line:4, col:5> 'b' 'Integer'
  | `-ConstExpr <line:3, col:24> 'Integer' '3'
  `-PrintCall <line:5, col:5> 'PrintCall'
    |-ImplicitTypeCast <line:5, col:11> 'Ptr{char[6]}'
    | `-ConstExpr <line:5, col:11> 'char[6]' 'abcdef'
    `-ConstExpr <line:5, col:26> 'Integer' '123'
```

Листинг 9 – Результат свёртки констант

5.2 Устранение тривиальных алгебраических выражении

Тестирование устранения тривиальных алгебраических выражении произовдилось на примере программы представленного на листинге №10. Результат выполнения данной оптимизации, представленный на листинге №11, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
Sub F(a%)
b% = 0 + (a% + 0)
c% = 1 * (a% * 1)
d% = a% * 0
e# = a% / 1
End Sub
```

Листинг 10 – Пример для оптимизаций устранения тривиальных алгебраических выражении

```
`-SubroutineDef <line:1, col:1> 'F' 'void(Integer)'
  |-Variable <line:1, col:7> 'a' 'Integer'
  `-CompoundStatement <line:2, col:5>
    |-VariableDecl <line:2, col:5> 'b' 'Integer'
    | `-Variable <line:2, col:15> 'a' 'Integer'
    |-VariableDecl <line:3, col:5> 'c' 'Integer'
    | `-Variable <line:3, col:15> 'a' 'Integer'
    |-VariableDecl <line:4, col:5> 'd' 'Integer'
    | `-ConstExpr <line:4, col:15> 'Integer' '0'
    `-VariableDecl <line:5, col:5> 'e' 'Double'
      `-ImplicitTypeCast <line:5, col:10> 'Double'
        `-Variable <line:5, col:10> 'a' 'Integer'
```

Листинг 11 – Результат свёртки констант

5.3 Оптимизация графа потока управления

Тестирование оптимизаций графа потока управления произовдилось на примере программы представленного на листинге №12. Результат выполнения данной оптимизации, представленный на рисунке №4, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
Function Join$()
  For i% = 2 To 10
    If i% = 10 Then
      Exit for i%
    End If
  Next i%
End Function
```

Листинг 12 – Пример для оптимизаций графа потока управления

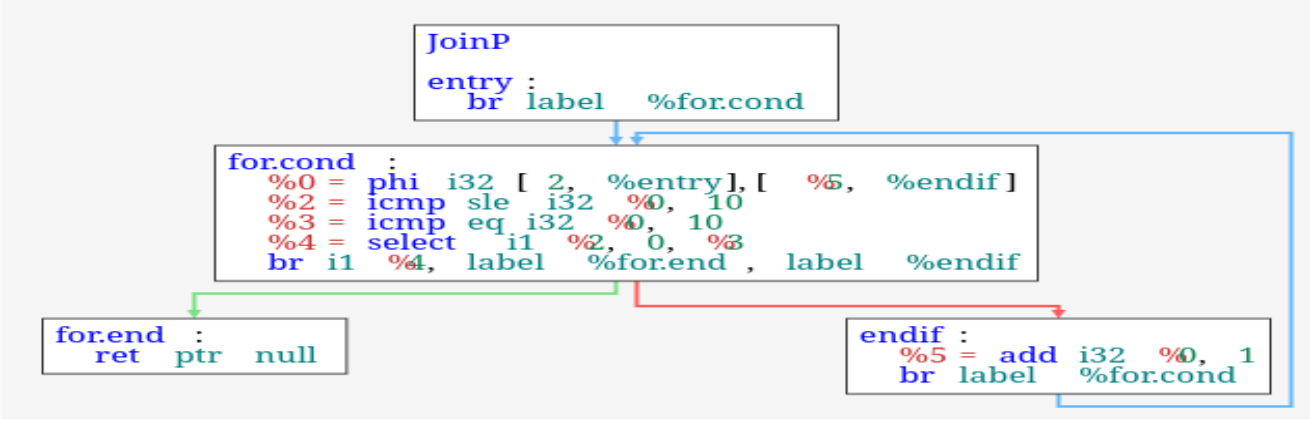

Рисунок 4 - Пример BasicHIR после упрощения ветвлении

5.4 Скалярное замещение

Тестирование оптимизаций скалярного замещения произовдилось на примере программы представленного на листинге №12. Результат до и полсе

выполнения данной оптимизации, представленный на листинге №4 и листинге №5, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

5.5 Удаление мёртвого кода

Тестирование оптимизаций удаления мёртвого кода произовдилось на примере программы представленного на листинге №13. Результат до и полсе выполнения данной оптимизации, представленный на листинге №14 и листинге №15, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
Sub F(a%)
   b% = 0 + (a% + 0)
   c% = 1 * (a% * 1)
   d% = a% * 0
   e# = a% / 1
   Print e#
End Sub
```

Листинг 13 – Пример для оптимизаций удаления мёртвого кода

```
define void @F(i32 %aI){
entry:
  %0 = add i32 %aI, 1
  %1 = add i32 %0, 1
  %2 = add i32 %1, 1
  %3 = sitofp double %aI
  call void @PrintD(double %3)
  ret void
}
```

Листинг 14 – Пример до оптимизаций удаления мёртвого кода

```
define void @F(i32 %aI){
entry:
  %3 = sitofp double %aI
  call void @PrintD(double %3)
  ret void
}
```

Листинг 15 – Пример после оптимизаций удаления мёртвого кода

5.6 Распространение констант

Тестирование оптимизаций распространения констант произовдилось на примере программы представленного на листинге №16. Результат до и полсе

выполнения данной оптимизации, представленный на листинге №17 и листинге №18, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
Sub Main(args$())
    a% = 10 * 20
    b% = 30 * 40
    d% = a% + b%
    e% = 15 + d%
    Print e%
end SUB
```

Листинг 16 – Пример для оптимизаций распространения констант

```
define void @Main(ptr %argsP){
entry:
  %0 = add i32 200, 1200
  %1 = add i32 15, %0
  call void @PrintI(i32 %1)
  ret void
}
```

Листинг 17 – Пример до оптимизаций распространения констант

```
define void @Main(ptr %argsP){
entry:
  call void @PrintI(i32 1415)
  ret void
}
```

Листинг 18 – Пример после оптимизаций распространения констант

5.7 Глобальная нумерация значений

Тестирование глобальной оптимизаций нумерации значении произовдилось на примере программы представленного на листинге №19. Результат до и полсе выполнения данной оптимизации, представленный на листинге №20 и листинге №21, показывает корректность данной трансформации исходного текста и улучшает производительность программы.

```
function foo%(x%, y%)
    a% = 1
    b% = 1
    do
        x% = x% + y%
        a% = a% + x%
        b% = b% + x%
    loop while x% > 0
    foo% = x% + b%
end function
```

Листинг 19 – Пример для глобальной оптимизаций нумерации значении

```
define i32 @fooI(i32 %xI, i32 %yI){
entry:
  br label %while.body

while.body:
  %0 = phi i32 [1, %entry], [%7, %while.body]
  %1 = phi i32 [1, %entry], [%6, %while.body]
  %3 = phi i32 [%xI, %entry], [%5, %while.body]
  %5 = add i32 %3, %yI
  %6 = add i32 %1, %5
  %7 = add i32 %0, %5
  %8 = icmp sgt i32 %5, 0
  br i1 %8, label %while.body, label %while.end

while.end:
  %9 = add i32 %5, %7
  ret i32 %9
}
```

Листинг 20 – Пример до глобальной оптимизаций нумерации значении

```
define i32 @fooI(i32 %xI, i32 %yI){
entry:
  br label %while.body

while.body:
  %0 = phi i32 [1, %entry], [%6, %while.body]
  %3 = phi i32 [%xI, %entry], [%5, %while.body]
  %5 = add i32 %3, %yI
  %6 = add i32 %0, %5
  %8 = icmp sgt i32 %5, 0
  br i1 %8, label %while.body, label %while.end

while.end:
  %9 = add i32 %5, %6
  ret i32 %9
}
```

Листинг 21 – Пример после глобальной оптимизаций нумерации значении

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы был разработан оптимизирующий компилятор для диалекта языка Basic, включающий ряд современных техник оптимизации, направленных на повышение эффективности и производительности сгенерированного кода. Были реализованные следующие методы оптимизации – свёртка констант, устранение тривиальных алгебраических выражении, упрощение графа управления, скалярное замещение, распространение констант, удаление мёртвого кода и глобальная нумерация значений.

Реализованные алгоритмы оптимизации продемонстрировали значительное улучшение эффективности генерируемого кода. Сокращение числа операций, уменьшение размера кода и улучшенное использование аппаратных ресурсов позволили достичь более высокой производительности программ, написанных на диалекте языка Basic.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. К63 Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2008. — 1184 с. : ил. — Парал. тит. англ. 15BM 978-5-8459-1349-4 (рус.)

[2] LLVM: LLVM Language Reference: [сайт]. – По всему миру, 2024. – URL: https://llvm.org/docs/LangRef.html (дата обращения 27.05.2025). – Текст. Изображение: электронные.

[3] Владимиров К. В57 Оптимизирующие компиляторы. Структура и алгоритмы – Москва: Издательство АСТ, 2024. – 272с. — ISBN 978-5-17-167965-1