



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

ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ НА ТЕМУ:

*«Улучшение результатов суперкомпиляции SCP4
посредством использования промежуточного
интерпретатора»*

Студент ИУ9-82

(Подпись, дата)

Атымханова М.

Руководитель ВКР

(Подпись, дата)

Коновалов А.В.

Консультант

(Подпись, дата)

(И.О.Фамилия)

Консультант

(Подпись, дата)

(И.О.Фамилия)

Нормоконтролер

(Подпись, дата)

(И.О.Фамилия)

Москва, 2019 г.

АННОТАЦИЯ

Объем данной дипломной работы составляет 50 страниц. Для ее написания было использовано 11 источников. В работе содержатся 41 листинг и 7 рисунков.

В дипломную работу входят пять глав. В первой главе описывается постановка задачи, во второй главе содержится описание предметной области, а оставшиеся три посвящены разработке, реализации и тестированию.

Объектом исследований данной ВКР является суперкомпилятор SCP4.

Цель работы — улучшение результатов суперкомпиляции SCP4. Поставленная цель достигается за счет разработки и реализации интерпретатора для языка входных графов SCP4. Интерпретатор реализован пригодным для его суперкомпиляции SCP4.

В работе показано, что специализация интерпретатора для некоторых исходных программ при помощи SCP4 способна давать лучшие остаточные программы, чем непосредственная суперкомпиляция этих программ.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Постановка задачи	6
2 Суперкомпиляция языка Рефал	7
2.1 Понятие метасистемного перехода в компьютерных науках	7
2.2 Понятия суперкомпиляции	9
2.3 Проекция Футамуры	14
2.3.1 Первая проекция Футамуры	15
2.3.2 Вторая и третья проекции Футамуры	17
2.4 Суперкомпилятор SCP4	17
2.4.1 Руководство пользователя для SCP4	17
2.4.2 Архитектура	19
2.5 Рефал	21
2.6 MST-схемы	23
2.6.1 Понятие входного Рефал-графа	26
3 Разработка интерпретатора	27
3.1 Входной граф для SCP4	27
3.1.1 Грамматика языка, на котором описывается промежуточное графовое представление функции программы	27
3.2 Интерпретатор графов, пригодный для суперкомпиляции	29
3.2.1 Режимы работы	29
3.2.2 Алгоритм работы	31
4 Реализация интерпретатора	33
5 Тестирование	43

5.1	Исследование программ, где применение интерпретатора порождает более интересные остаточные программы	43
	ЗАКЛЮЧЕНИЕ	47
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	48
	ПРИЛОЖЕНИЕ А	50

ВВЕДЕНИЕ

Суперкомпиляция (supervising compilation) — это метод преобразования программ с целью их оптимизации и анализа. Такой метод использует технику преобразования программ, основанную на построении полной и самодостаточной модели программы. Процедура суперкомпиляции заключается в «наблюдении» модели исполнения программы. Суперкомпиляция программы порождает другую программу эквивалентную как функция исходной, но с измененной операционной семантикой.

Используемый в данной выпускной квалификационной работе (далее, ВКР) суперкомпилятор — это суперкомпилятор SCP4 (далее, SCP4) [1] для языка РЕФАЛ-5.

Основная цель данной ВКР — изучить формат ранее неопisanного входного в SCP4 графового представления программы, улучшить результаты суперкомпиляции SCP4. Стратегия улучшения — написание интерпретатора для входного в SCP4 представления программы, который будет просуперкомпилирован. Суперкомпиляция интерпретатора, выполняющего исходную программу, должна порождать лучшую версию остаточной программы, чем просто суперкомпиляция исходной.

Как результат ожидается успешная суперкомпиляция интерпретатора по заданной программе, что в теории становится скомпилированной программой на выходной язык, и порожденная программа будет иметь преимущества по выполнению и сложности над изначальной программой и преимущества над программой порожденной суперкомпиляцией исходной.

1 Постановка задачи

1. Изучение теории суперкомпиляции языка Рефал

- (a) Изучение понятий суперкомпиляции
- (b) Изучение аспектов суперкомпиляции Рефала: понятие Рефал-графа, понятие входного Рефал-графа
- (c) Изучение проекций Футамуры

2. Написание интерпретатора графа

- (a) Изучение суперкомпилятора SCP4 : пользовательский интерфейс, архитектура
- (b) Изучение MST-схем : синтаксис, семантика, использование в SCP4
- (c) Изучение синтаксиса входных графов SCP4 , порождаемых проходом inref4
- (d) Написание интерпретатора графов, пригодного для суперкомпиляции

3. Эксперименты с суперкомпиляцией интерпретатора

- (a) Исследование примеров программ, где применение интерпретатора дает более интересные остаточные программы
- (b) Объяснение причин почему это работает: неявный учет окрестностей, сохранение информации об успешных отождествлениях

2 Суперкомпиляция языка Рефал

2.1 Понятие метасистемного перехода в компьютерных науках

В. Ф. Турчин в книге «Феномен науки» [2] сформулировал идею метасистемного перехода. Идея заключалась в переходе от системы объектов с управляющим элементом над ними к метасистеме, в которой управляющий элемент системы становится объектом для нового управляющего уровня (метапрограмма). Автор приводит в пример математику, в которой понятие алгебры является метасистемным переходом над арифметикой.

Целью метасистемного перехода в программировании является переход от написания программ к написанию метапрограмм, которые сами анализируют входные программы и порождают новые.

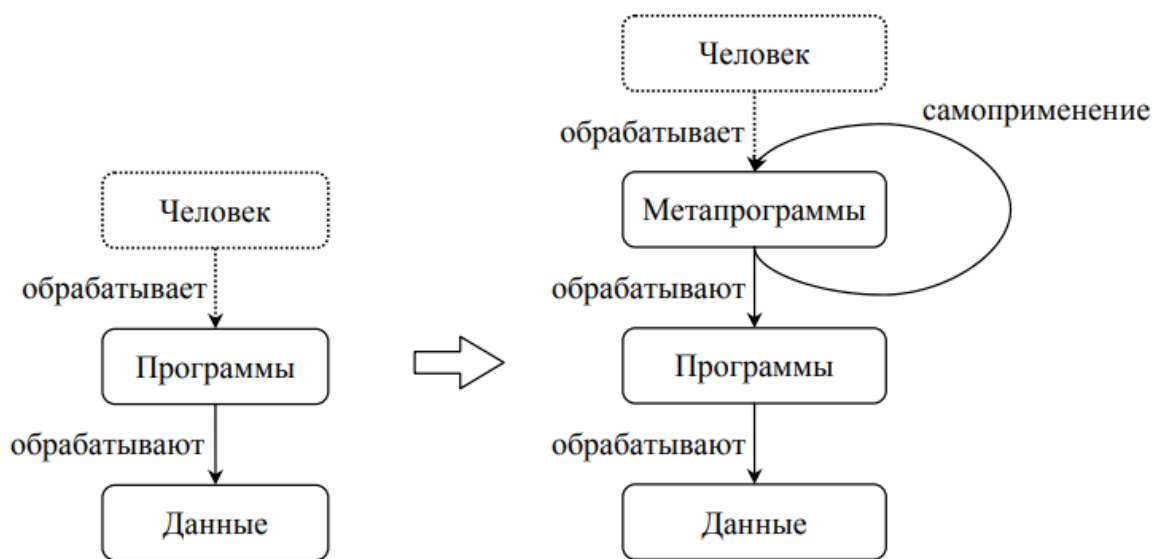


Рис. 1: Метасистемный переход от программирования как деятельности человека над программами к автоматизированной и автоматической обработке программ на компьютере и метапрограммированию [3]

Основными задачами метавычислений (деятельностью по преобразованию программ и метапрограмм) становятся:

1. Специализация программ — задача порождения по программе ее модифи-

цированной версии с фиксацией некоторой части параметров. Известная информация преобразует весь текст программы и делает ее (программу) более пригодной для оптимизаций. После чего, вычисление для остаточной программы становится намного эффективнее.

$$f(x, y) \rightarrow f_A(y) = f(A, y)$$

2. Композиция программ — это порождение по двум (или нескольким) подпрограммам, передающим результаты одной в качестве аргументов другой, более эффективной программы, которая либо выполняет ту же работу вообще без формирования промежуточных данных («за один проход»), либо содержит прооптимизированные версии исходных подпрограмм с учетом того, что они работают в контексте друг друга [3].

$$f(x), g(x) \rightarrow f_g(x) = f(g(x))$$

Композиция программ является обобщением специализации, поскольку

$$f_A(y) = f(g(y), y), \quad g(y) = A$$

3. Инверсия программ — порождение по программе новой программы, которая выдает для данных, являющихся результатом работы первой программы в качестве ответа аргумент первой программы для которой получался ответ, ставший входными данными для инверсии. Зачастую на практике задать обратную функцию легче чем реализовать прямую функцию. И инверсионная задача для обратной будет прямой. Ее порождение выполнит такой метаинструмент.

$$f(x) \rightarrow f^{-1}(y) = x, \quad y = f(x)$$

Эти задачи метавычислений берется решать метапрограмма. Такой метапрограммой стал суперкомпилятор, реализующий идеи суперкомпиляции, которая была предложена В.Ф Турчиным.

2.2 Понятия суперкомпиляции

Программа рассматривается в виде некоторой машины. Смысл извлекается из наблюдения за ее действиями. Суперкомпилятор не преобразует программу шаг за шагом; он управляет и наблюдает за машиной (M_1), которая представлена в виде программы. Наблюдая за работой машины M_1 суперкомпилятор конструирует другую машину M_2 , которая описывает действия машины M_1 . При построении M_2 суперкомпилятор пользуется различными трюками, чтобы M_2 работала так же, как и M_1 , но быстрее. Цель суперкомпилятора – сделать M_2 самодостаточной. По достижении этой цели суперкомпилятор отбрасывает исходную машину M_1 и выдаёт M_2 . Суперкомпилятор запускает M_1 в общем виде (с неизвестными значениями переменных), строит граф состояний и переходов между различными конфигурациями вычислительной системы. Такой граф полностью описывает поведение системы. Таким образом, новая программа становится самодостаточной моделью исходной программы[4].

Идея суперкомпиляции:

1. Программе P_1 ставится в соответствие машина M_1 , моделирующая в общем виде выполнение программы P_1 .
2. Контролируя и наблюдая (SUPERvises) работу машины M_1 , суперкомпилятор создает (компилирует, COMPILEs) другую машину M_2 , которая полностью описывает M_1 .
3. Машина M_2 далее может быть представлена в виде программы P_2 [5].

Алгоритм суперкомпиляции:

1. Программа выполняется символически для всех возможных аргументов, удовлетворяющих каким-то ограничениям. Для этого строится «дерево конфигураций» (= «дерево процессов»). Узлы дерева — «конфигурации», они описывают множества состояний вычислительного процесса. Переходы

между узлами соответствуют каким-то действиям и проверкам, происходящим при исполнении программы.

2. Дерево конфигураций получается бесконечным, если исходная программа содержит циклы и/или рекурсию. В процессе суперкомпиляции бесконечное дерево сворачивается в конечный «граф конфигураций». Для этого конфигурации сравниваются между собой (отношение Турчина, отношение Хигмана-Крускала). Если появляются две похожие конфигурации, делается попытка «свести» ту конфигурацию, что находится в дереве ниже, к той, что появилась выше. В результате в дереве появляется «обратная» стрелка, и дерево превращается в граф с циклами [6].

Введём основные понятия на примере суперкомпиляции программы с композицией функций Fab, которая заменяет во входном аргументе встречающиеся буквы «a» на «b», и Fbc, которая заменяет во входном аргументе встречающиеся буквы «b» на «c».

Листинг 1: Определение функции Fab на языке Рефал

```
Fab{  
  'a' e.X = 'b' <Fab e.X>;  
  s.K e.X = s.K <Fab e.X>;  
  /* empty */ = /* empty */;  
}
```

Листинг 2: Определение функции Fbc на языке Рефал

```
Fbc{  
  'b' e.X = 'c' <Fbc e.X>;  
  s.K e.X = s.K <Fbc e.X>;  
  /* empty */ = /* empty */;  
}
```

Определение функции на языке Рефал состоит из названия функции и блока с предложениями. Каждое предложение содержит две части. Левая на-

зывается образцовым выражением, а правая результатным. Образец состоит из записи в составе которой присутствуют разные термы и переменные языка Рефал. При исполнении программы Рефал-машиной вызов функции с одним аргументом выполняется следующим образом. Аргумент декомпозируется для сопоставления с образцом первого предложения. Если сопоставление происходит успешно, то в поле зрения Рефал машины вызов функции заменяется результатным выражением, для которого сопоставление с образцом прошло успешно. Сопоставление осуществляется с образцом каждого предложения в порядке записи в определении функции до успешного сопоставления. Вызов функции заключается в структурные скобки $\langle FuncName \quad arg \rangle$. Этой информации о Рефале достаточно для понимания небольшого примера суперкомпиляции программы на Рефале. В дальнейших главах синтаксис и семантика языка Рефал будут раскрыты подробнее.

Итак, для суперкомпиляции композиции функций изначально в поле зрения суперкомпилятора находится выражение

$$\langle Fbc \langle Fab \quad e.0 \rangle \rangle$$

$e.0$ — аргумент для функции Fab и $\langle Fab \quad e.0 \rangle$ — аргумент для функции Fbc . Но для суперкомпилятора $e.0$ является параметром, над которым он проводит символьные вычисления.

Для текущего поля зрения суперкомпилятор пытается лениво вычислить значение вызова функции $\langle Fbc... \rangle$. Но так как аргумент содержит вызов функции, суперкомпилятор помещает вызов функции из аргумента на вершину стека, указывая что аргументом вызова первичной функции становится еще не вычисленное значение на вершине стека. Формируется следующий стек вызовов функций:

$$\langle Fbc \langle Fab \quad e.0 \rangle \rangle \rightarrow \begin{array}{l} \langle Fab \quad e.0 \rangle \leftarrow h.0 \\ \langle Fbc \quad h.0 \rangle \end{array} \quad (1)$$

Из текущего состояния осуществляется спекулятивное выполнение с разными возможными значениями параметров, которое называется прогонкой (Рисунок 2,3). При обычных вычислениях начальное состояние определяется конкретными значениями. Но в прогонке мы оперируем символами переменных и в зависимости от значения переменной процесс вычисления должен перейти на одну или другую ветвь.

Считывая на каждом шаге состояние вычислительного устройства, мы наблюдаем процесс вычислений и можем записать его трассу, путь.

В узлах построенных деревьев находятся состояния вычисления программы, в которых демонстрируется стек вычислений, а переходы содержат информацию о возможном сопоставлении аргумента с образцом и следующую из этого конфигурацию.

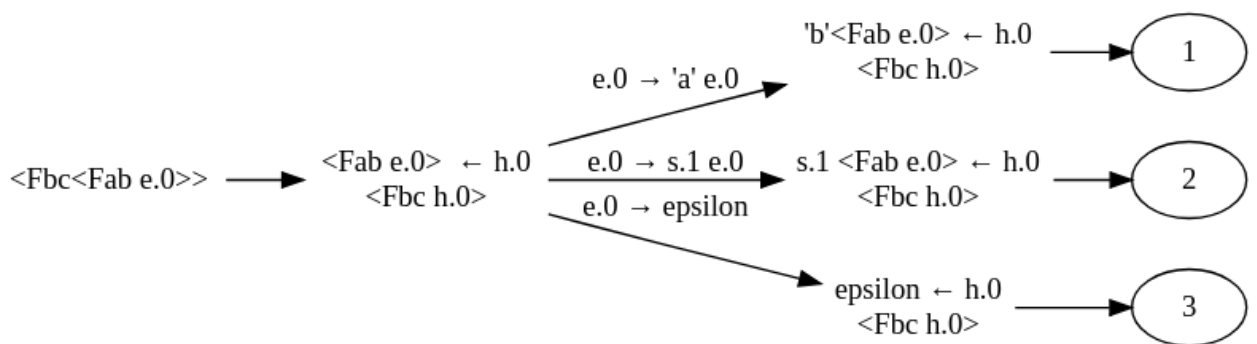


Рис. 2: Ветка прогонки

На Рис.2 можно проследить ветвление процесса символического вычисления программы. Результату исполнения программы в дереве всегда соответствует один путь.

На Рис.3 продолжение «прогонки».

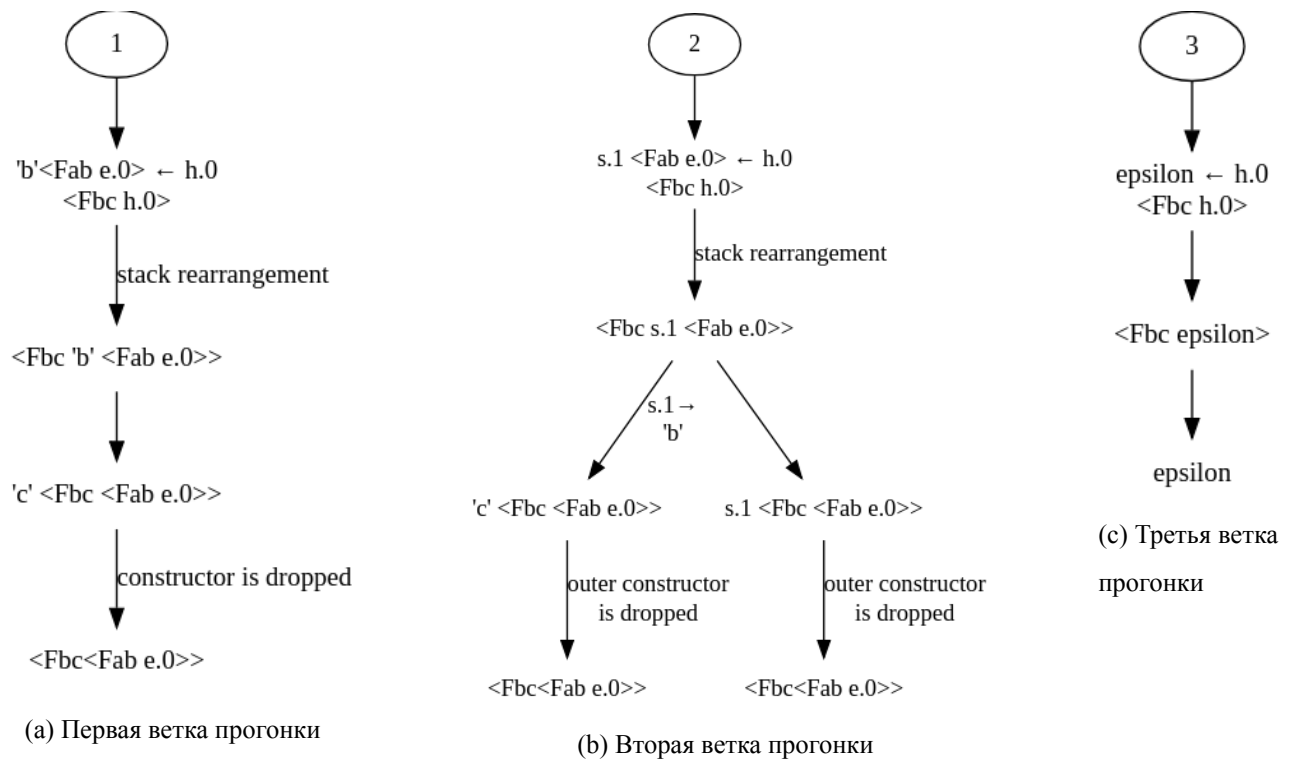


Рис. 3: Участки дерева процесса

Три из четырех листов дерева содержат начальную конфигурацию (а четвертый лист содержит конечный результат), что позволяет не продолжать прогонку, а провести обратную стрелку в вершину дерева. Таким образом, бесконечное дерево преобразовалось в конечный граф с циклами (Рис. 4). По нему

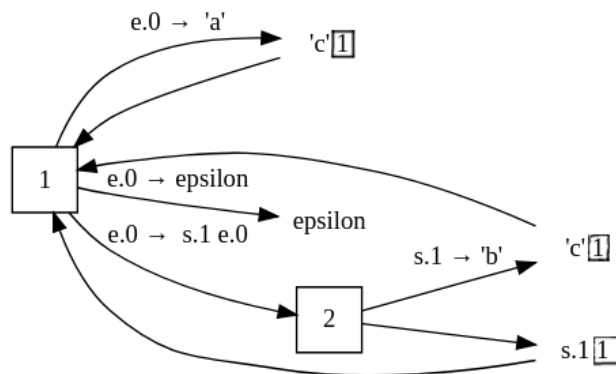


Рис. 4: Дерево превратилось в граф

был порожден остаточный граф — Рис.5.

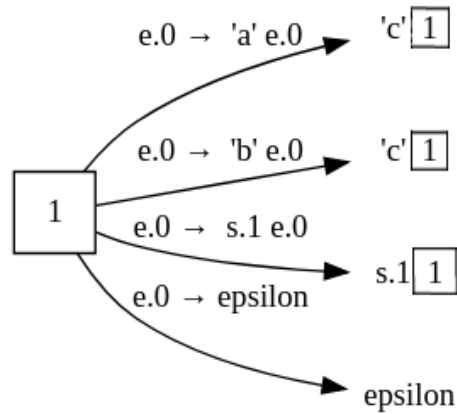


Рис. 5: Конечный граф

Каждый базисный узел графа кодогенерирует функцию. В рассматриваемом примере остаточная программа получилась следующей (Листинг 3). Суперкомпилятор решил задачу композиции для функций Fab и Fbc . На этом примере продемонстрирован алгоритм прогонки.

Листинг 3: Остаточная программа

```
F1{
  'a' e.0 = 'c' <F1 e.0>;
  'b' e.0 = 'c' <F1 e.0>;
  s.K e.X = s.K <F1 e.X>;
  /*epsilon*/ = /*epsilon*/;
}
```

2.3 Проекция Футамуры

Суперкомпиляция является методом, который может применяться для решения различных задач. Одной из таких задач является специализация программ.

Абстрактно задачу, решаемую специализатором программ, можно описать следующим образом.

Пусть P — исходная программа, а C — ограничения на условия эксплуата-

ции P . Тогда на вход специализатора подается (P, C) , а задача специализатора — породить остаточную программу P' , удовлетворяющую следующим требованиям.

- P' эквивалентна P , если выполнены условия C .
- P' получается путем устранения из P тех частей и действий, которые становятся ненужными, в результате наложения условий C .

Надежды и ожидания, связанные со специализацией P , состоят в следующем.

- P' будет эффективнее, чем P .
- P' будет меньше, чем P .
- P' будет проще, чем P .

2.3.1 Первая проекция Футамуры

Идея первой проекции в следующем[7]. Будем полагать, что программы у нас — чистые функции, которые пишутся на некотором функциональном языке. Пусть у нас имеется программа $P_R(d1, d2)$, которая принимает два аргумента. Индекс R подразумевает, что программа написана на языке R . Для каждого языка программирования мы подразумеваем, что есть некоторая машина (вымышленная), которая может выполнять программы на этом языке.

Эту машину у программистов принято называть интерпретатором, а у математиков универсальной функцией.

И пусть у нас есть специализатор. $d1$ — конкретное значение.

$$SPEC_R(P_R, d1) = P_{R,d1}$$

Он порождает программу

$$P_{R,d1}(d2') \equiv P_R(d1, d2')$$

— функцию одного аргумента $d2$, которая для любых значений $d2'$ вычисляет то же значение, что и исходная программа $P_R(d1, d2')$ при конкретном значении $d1$. Можно записать формулу

$$SPEC_R(P_R, d1)(d2) = P_R(d1, d2)$$

, где $d1$ — константа, $d2$ — параметр.

Интерпретатор $INTS_R(P_S, d)$ — программа на языке R , которая принимает текст программы на языке S , и входное данное для этой программы, и возвращает тот же результат, что и непосредственное выполнение $P_S(d)$ на машине языка S . Т.е. при помощи интерпретатора мы можем на машинах R выполнять программы для машин S , не имея при этом машины S .

$$[[INTS_R(P_S, d)]] = [[P_S(d)]]$$

А теперь вернёмся к предыдущей формуле и подставим вместо P_R — $INTS_R$, вместо $d1$ — P_S :

$$SPEC_R(INTS_R, P_S)(d) = INTS_R(P_S, d) = P_S(d)$$

(Не обязательно, чтобы специализатор $SPEC$ был написан на том же языке, что и интерпретатор)

Что такое $SPEC(INTS_R, P_S)$? Это некоторая программа на языке R (для машины R):

$$INTS_{R, P_S}(d') = INTS_R(P_S, d') = P_S(d')$$

Значит, интерпретатор, специализированный по интерпретируемой программе — это программа для машины R , которая для любых входных данных d' выдаёт тот же результат, что и программа P_S , запущенная на машине S . Т.е. результат специализации интерпретатора — это программа P_S скомпилированная для машины R :

$$P_R(d') \equiv INTS_{R, P_S}(d') = INTS_R(P_S, d') = P_S(d')$$

2.3.2 Вторая и третья проекции Футамуры

Сам специализатор является программой на языке R от двух аргументов. А значит, его тоже можно специализировать:

$$\begin{aligned} P_S(d) &= INTS_R(P_S, d) \\ &= SPEC_R(INTS_R, P_S)(d) = \\ &SPEC_R(SPEC_R, INTS_R)(P_S)(d) \\ &= SPEC_R(SPEC_R, SPEC_R)(INTS_R)(P_S)(d) \end{aligned} \tag{2}$$

(Специализировать можно по разным аргументам и даже по части аргумента. Здесь специализация по первому аргументу. Т.е. тому, который полностью известен. Второй аргумент является параметром.)

Вторая проекция:

Программа, которая принимает программу на языке S и формирует эквивалентную программу на языке R — это компилятор.

$$\begin{aligned} P_R &= SPEC_R(INT_R, P_S) = SPEC_R(SPEC_R, INT_R)(P_S) = COMP_{R \rightarrow S}(P_S) \\ SPEC_R(SPEC_R, INT_R) &= COMP_{R \rightarrow S} \end{aligned}$$

Третья проекция — компилятор компиляторов, программа, которая превращает интерпретаторы в компиляторы:

$$\begin{aligned} COMP_{R \rightarrow S} &= SPEC_R(SPEC_R, INT_R) \\ &= SPEC_R(SPEC_R, SPEC_R)(INT_R) = COMCOM_R(INT_R) \end{aligned} \tag{3}$$

2.4 Суперкомпилятор SCP4

2.4.1 Руководство пользователя для SCP4

Инструментом выполняющим роль специализатора (и не только) в этой ВКР становится SCP4.

Листинг 4: Инструкция по сборке 1

```
$ wget https://mazdaywik.github.io/direct-link/  
scp4new-version-05-03-2019-srefc.zip -O scp4.zip  
  
$ unzip scp4.zip
```

Листинг 5: Инструкция по сборке 2

```
~/scp4/$ srefc install  
~/scp4/$ ./install linux  
~/scp4/$ ./refcall.bat  
~/scp4/$ rm *.rasl
```

Суперкомпилятор принимает два аргумента: программу и задание на суперкомпиляцию. Это могут быть два отдельных файла `file.ref` (программа) и `taskforsupercmp.mst` (задание). Для этих файлов в наборе файлов суперкомпилятора отводится папка `scp4/test/`.

Листинг 6: Инструкция по запуску

```
~/scp4/test$ export PATH=$PATH:$(pwd)/..  
~/scp4/test$ scpwholp file.ref taskforsupercmp.mst
```

Также в программе можно добавить псевдокомментарий:

```
*$MST_FROM_ENTRY;
```

т.е. комментарий в коде, который суперкомпилятором не игнорируется, а учитывается. И он формирует MST-схему вида

$$\begin{aligned} < F \quad e.0 > \\ < G \quad e.0 > \\ \dots \end{aligned} \tag{4}$$

где функции F, G, \dots были объявлены как `$ENTRY`.

В случае следующего примера заданием на суперкомпиляцию будет функция, являющаяся входной точкой программы на Рефале — Go .

Листинг 7: Задание на суперкомпиляцию в программе *.ref*

```
*$MST_FROM_ENTRY;  
  
$ENTRY Go {  
    e.x = <Prout Hello>;  
    /*empty*/=/*empty*/;  
}  
...
```

Для такой программы запуск осуществляется так:

Листинг 8: Инструкция по запуску без файла с заданием

```
~/scp4/test$ export PATH=$PATH: 'pwd' /..  
~/scp4/test$ scpwholp file.ref
```

В результате выполнения этих команд будет сгенерирован файл на языке Рефал-5 «*r_taskforsupercmp.ref*» с программой, порожденной суперкомпилятором.

2.4.2 Архитектура

Для того, чтобы осуществить суперкомпиляцию над программой на Рефале с использованием SCP4 после запуска из раздела «Руководство пользователя для SCP4» программа пройдет через следующие шаги:

1. Программа на входном языке попадает на преобразование суперкомпилятора в подготовленном виде. Компилятор *inref4* преобразует входную программу **.ref* в множество файлов, где каждой функции соответствует ее граф промежуточного представления. Эти файлы используются суперкомпилятором SCP4.
2. Также суперкомпилятор SCP4 принимает скомпилированное задание на суперкомпиляцию на языке MST-схем. О формулировке задания подробнее в

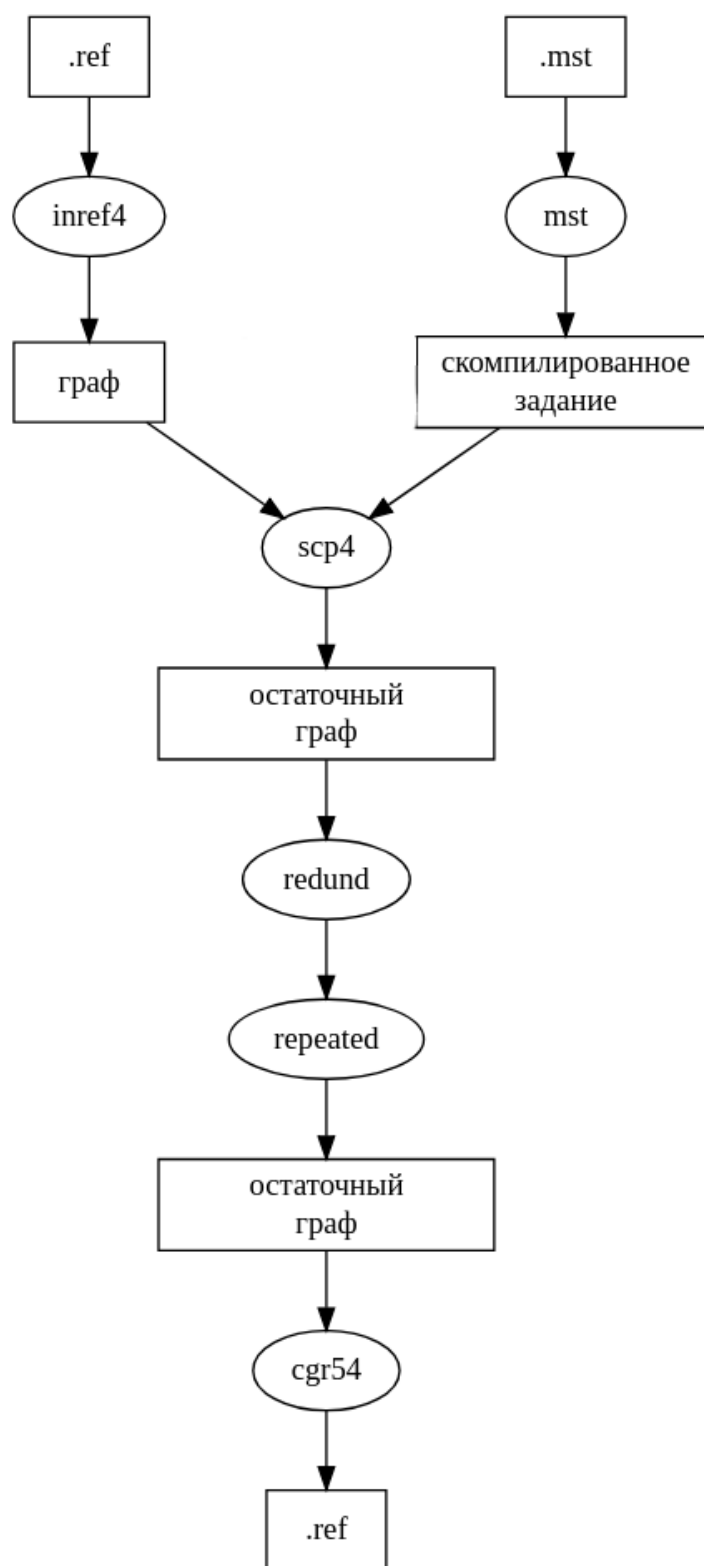


Рис. 6: Конвейер суперкомпиляции программы и задания

дальнейших главах.

3. Скомпилированные программа и задание становятся входными данными

для суперкомпилятора SCP4. Из задания на суперкомпиляцию и множества файлов входных графов SCP4 конструирует бесконечное дерево конфигураций, которое затем сворачивается в граф конфигураций с применением техник прогонки, обобщения, встраивания.

4. К полученному графу применяются оптимизации устранения неиспользуемых функций и удаления повторных переменных.
5. Компилятор из графового представления программы в Рефал-5 *crg54* порождает по остаточному графу программу на Рефале-5.

2.5 Рефал

РЕФАЛ (РЕкурсивных Функций АЛгоритмический язык) — это функциональный язык программирования, ориентированный на так называемые «символьные преобразования»: обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом.

РЕФАЛ-5 является диалектом языка РЕФАЛ, который был разработан в Нью - Йоркском Сити Колледже [8].

Данные в Рефале[9]

1. Символы

- (a) '+' — символ-литера (character)
- (b) Alpha-2 — составной символ, имя функции
- (c) 125 — символ-число

2. Структурные скобки (...)

3. Терм

- (a) Символ

(b) Выражение в скобках

4. Выражение

(a) Последовательность термов

Управление в Рефале

1. Вызов функции

<Имя функции Выражение>

<Fact 5> — пример

Поместить начальное выражение в поле зрения РЕФАЛ-машины можно следующим способом. РЕФАЛ-5 использует соглашение, согласно которому начальным состоянием поля зрения всегда является:

<Go>

Это значит, что среди функций, определенных в выполнимой программе, всегда должна быть стандартная функция Go без аргументов:

Листинг 9: Функция Go — входная точка программы

```
$ENTRY Go {  
    = the initial Refal expression  
}
```

2. Определение функции [10]

Имя функции {

образец = результат ; /* предложение */

...

}

Программа на Рефале может состоять из одного или нескольких модулей (файлов), каждый из которых, в свою очередь, состоит из функций.

Рефал-функция представляет собой упорядоченный набор предложений, состоящих из образца и результатного выражения. На вход функции подаётся некоторое выражение; вычисление функции состоит в сопоставлении выражения поочередно образцам, взятым из первого, второго и т. д. предложений. Если очередное сопоставление проходит успешно, то на основании результатного выражения, взятого из того же предложения, формируется новое Рефал-выражение, которое и будет результатом вызова функции.

3. Переменные (первый символ s , t , e — тип переменной)

- (a) $s.X$ — произвольный символ
- (b) $t.Y$ — произвольный терм
- (c) $e.l$ — произвольное выражение

Читателю в дальнейшем понадобятся знания о конструкциях в Рефале для понимания раздела «Тестирование».

2.6 MST-схемы

На языке MST-схем формулируется задание на суперкомпиляцию. То есть задание на специализацию программы по ее данным. В данной ВКР этой программой будет интерпретатор, а данными для него — программа на языке входных Рефал-графов.

В основе лежит философия метасистемного перехода: «перехода от системы, состоящей из некоторых объектов и управляющей системы, осуществляющей какую-то деятельность над этими объектами, управление ими, к метасистеме, в которой возникает новый уровень управления, для которого предшествующий уровень становится объектом деятельности[2]».

Суперкомпиляция как метод осуществления метасистемного перехода подходит для реализации метасистемного перехода над алгоритмами, программа-

ми, формальными системами, т.е. программы становятся объектами для глубоких преобразований.

В данной ВКР будет производиться наблюдение(с помощью суперкомпилятора) за поведением программы-интерпретатора, написанной на Рефале.

Рефал является языком первого уровня. Мы не можем манипулировать над программами на Рефале программой написанной на Рефале, так как входная программа на Рефале может содержать вызовы функций, которые будут вычисляться при манипуляции над входной программой, и открытые переменные. Поэтому следует отобразить некоторые структуры программы на Рефале в объектные выражения для манипуляции внешней Рефал программой. Формализация такого задания демонстрируется следующим листингом на языке MST-схем:

Листинг 10: Задание на суперкомпиляцию «task.mst»

```
$DEFINE Prog
<Inref4
  Fab {
    'a' e.X = 'b' <Fab e.X>;
    s.1 e.X = s.1 <Fab e.X>;
    = ;
  }
>;

<Int      e.X      (      ) > --
  <Go      !      >   %.Prog      ;
```

1. *\$DEFINE Prog...*; — объявление макроса Prog
2. *< Inref4 ... >* — вызов компилятора программы на Рефале в ее Рефал-граф
3. *Fab { /* definition */ }* — функция на языке Рефал, заменяющая во входном выражении символы 'a' на 'b'.

Макрос осуществляет препроцессинг входной программы записанной как аргумент в структурных скобках вызова компилятора в промежуточное представление. Эти данные становятся входными данными в специализации интерпретатора по программе.

Листинг 11: Многоуровневая запись задания

```
<Int      e.X      (      ) > --
      <Go    !    >    %.Prog    ;
```

Функция *Int* выполняется для входных данных представленных программой на уровень ниже *<Go ! >*, где *'!'* обозначает поднятие аргумента функции *Go*, то есть для функции *Go* он перестает быть аргументом и становится параметром. Специализация происходит по программе, для которой в макросе вызывался компилятор в промежуточное представление.

В полученной иерархии вызовов аргументы функций должны быть закодированы то количество раз, которое соответствует уровню в иерархии (счет ведется начиная с нижнего уровня). Входные данные для функции должны быть метакодированы по следующим правилам:

Метакод SCP4 [11]:

1. символы неизменны
2. в начало скобочных термов добавляется *'*'*
3. вызовы функций изображаются *('!')(Fn ...) ...)*
4. переменные — *(Var s.Mode s.Index)*

Вертикальное расположение слов программы соответствует уровням действия функций друг над другом. Вводя нумерацию «снизу-вверх», можно говорить что программа уровня *n* является данными для программы уровня *n+1*. Каждый уровень в вертикальной записи это осуществление метасистемного пе-

перехода. '!' означает поднятие данных из текущей позиции программы, поднятые/опущенные данные метакодированы.

Файл с расширением .mst компилируется при вызове суперкомпилятора для программы и задания на суперкомпиляцию.

2.6.1 Понятие входного Рефал-графа

Итак, в дереве процессов программы узлами являются состояния программы, переходы между узлами соответствуют каким-то действиям и проверкам, происходящим при исполнении программы. Такие действия и проверки описываются в файле порождаемом в используемом в данном ВКР компилятором `inref4` из входной программы в графовое представление функций программы которое описывает соответствие входных аргументов функции конфигурациям следующих возможных состояний программы. Входной Рефал-граф является представлением исходной программы, удобным для суперкомпиляции. В нём сопоставление с образцами преобразовано в дерево элементарных команд отождествления.

Входной рефал-граф содержит информацию о функции из программы на Рефале. Граф описывает специализированное поведение функции по ее аргументам. Такое описание содержит следующие смысловые конструкции:

1. Ветвление — когда программа имеет несколько вариантов дальнейшего поведения в зависимости от аргументов
2. Сужение — когда образец в левой части предложения функции задает шаблон для аргумента
3. Рестрикция — когда аргумент не должен иметь вид определенного образца
4. Конфигурация — правая часть предложения, которая выполняется для определенной ветки специализации

3 Разработка интерпретатора

Интерпретатор, написанный в рамках ВКР, распознает язык входных графов для SCP4. Он принимает граф входной функции. Грамматика языка такого представления не была задокументирована. Грамматика была полностью восстановлена в рамках ВКР, а семантика определяется написанным в ходе ВКР интерпретатором. По заданию и множеству входных графов для вызываемых функций интерпретатор может выполнить исходную программу на Рефале.

3.1 Входной граф для SCP4

3.1.1 Грамматика языка, на котором описывается промежуточное графовое представление функции программы

Порождается проходом `inref4`, компилятором из языка Рефал в входной Рефал-граф. Папка «`../pgraph/`» заполняется графами в виде сериализованных объектных предложений.

Листинг 12: Грамматика языка записана в расширенной форме Бэкуса-Наура (РБНФ) 1

```
t.Fork ::= '(Fork ' e.Branches ' )'
e.Branches ::= e.Branch +
e.Branch ::= '+ ' (s.Index e.Chain)
e.Chain ::= t.ChainItem
t.ChainItem ::=
    (Ct e.Contr) /* contraction*/
    | (Rs e.Restr) /*restriction*/
    | (Fork e.Branches) /*branching*/
    | (Conf e.Conf) /*configuration*/
    | (Def e.Def) /*predefined function*/

e.Contr ::= t.Var ':' e.Pattern
t.Var ::= '(Var ' s.Type ' s.Index)'
s.Type ::= 's' | 't' | 'e'
```

Листинг 13: Грамматика языка записана в расширенной форме Бэкуса-Наура (РБНФ) 2

```
s.Index ::= s.NUMBER
e.Pattern ::= t.PatTerm*
t.PatTerm ::= s.SYMBOL | t.Var | ('*' e.Pattern)
e.Restr ::= t.Sym '#' t.Sym
t.Sym ::= (Var 's' s.Index) | s.SYMBOL
e.Conf ::= e.Expr ':' t.Var
e.Def ::= (e.Expr ':' t.Var)
e.Expr ::= t.Term*
t.Term ::= s.SYMBOL | t.Var | ('*' e.Pattern) |
          | ('!' t.Call ':' t.Var)
t.Call ::= (Fn s.FunvName e.conf)
s.FuncName ::= s.WORD
```

Пример графа построенного для функции. Функция всегда возвращает слово, не начинающееся с предиката 'A B'.

Листинг 14: Функция

```
Func {
  A B e.X = Hello;
  e.X = e.X;
  /*empty*/=/*empty*/;
}
```

Построенный для нее граф:

Листинг 15: Граф

```
(Fork      '+'(1
                                (Ct (Var 'e'1 ) ': 'A (Var 'e'1 ))
                                (Ct (Var 'e'1 ) ': 'B (Var 'e'1 ))
                                (Conf (Hello ':' (Var 'e'0 ))))
      '+'(2
                                (Conf ((Var 'e'1 ) ': (Var 'e'0 )))))
```

Он содержит:

1. Ветвление, так как в определении функции находятся два предложения с разными образцами
2. В первой ветке содержится сужение: проверка входного выражения на соответствие образцовому выражению, начинающемуся с символа 'А'. Второе сужение на новом объектном выражении осуществляет проверку с образцом выражения, начинающегося на символ 'В'.
3. Конфигурация, присваивающая возвращаемой е-переменной слово 'Hello'
4. Конфигурация для второй ветки (для образца, не содержащего предиката 'AB'), присваивающая возвращаемой е-переменной входную е-переменную

3.2 Интерпретатор графов, пригодный для суперкомпиляции

3.2.1 Режимы работы

Написанный в данной ВКР интерпретатор применим в двух режимах:

1. Интерпретатор для интерпретации программ
2. Интерпретатор для экспериментов с суперкомпиляцией (в этом режиме интерпретатор является входными данными для суперкомпилятора)

Интерпретатор принимает программы на языке графового представления функций программы, порождаемом проходом inref4.

Ход работы интерпретатора в режиме 1:

1. Запуск интерпретатора для скомпилированной программы в промежуточное состояние компилятором inref4.
2. Считать граф функции Go, являющейся входной точкой
3. Рекурсивно считать и сохранить графы всех функций вызываемых из текущей

4. Запустить функцию с алгоритмом интерпретации на множестве полученного «словаря» имен функций исходной программы и их графов

Ход работы интерпретатора в режиме 2:

В этом режиме текст программы интерпретатора становится входными данными для суперкомпилятора

1. В задании на суперкомпиляцию описывается вызов функции Int со всеми начальными параметрами. Поэтому первым становится вызов функции Int с заданными флагами и сформированными графами для функций. Само задание тоже компилируется. То есть интерпретатор попадает в суперкомпилятор тоже в виде набора файлов со своим графовым промежуточным представлением.
2. Исполнение программы интерпретатором наблюдается суперкомпилятором. Интерпретатор суперкомпилируется.
3. SCP4 порождает файл на рефале с остаточной программой интерпретатора специализированного по программе. При невозникновении ошибок (в работе SCP4) должна получиться скомпилированная программа, которая была заданием на интерпретацию.

Интерпретатор выполняя программу оперирует в своем внутреннем представлении следующими компонентами:

- Таблица переменных
- Граф
- Информация для отката (стек)

Возможные конфигурации в интерпретируемом графе:

- Если узел графа имеет вид «сужение»(contraction), то из таблицы переменных выбирается сужаемая переменная и проверяется удовлетворяет ли она

сужению.

Если удовлетворяет —применяется сужение, это изменяет таблицу.

Иначе выполняется откат.

- Если узел графа имеет вид «ветвления» (forking), то берется первая ветвь, остальные кладутся в список отката.
- Если узел графа имеет вид «рестрикции»(restriction), то проверяется рестрикция, иначе выполняется откат.
- Если узел графа имеет вид «конфигурация»(configuration), то подставляются переменные в выражение, выбирается вызов функции. Если вызов функции присутствует, то он выполняется и рекурсивно вызывается интерпретатор. Иначе возвращается значение выражения.
- «откат» —берется очередная ветвь Fork и выполняется.

3.2.2 Алгоритм работы

Алгоритм работы интерпретатора:

1. Интерпретатор принимает граф функции с заданием и «словарь» из имен функций и их графов, ключи стратегии вычисления(ленивый/ аппликативный), а также ключ с указанием типа веток к которым нужно применять текущую функцию.
2. Далее осуществляется синтаксический анализ графа функции и вычисляется соответствующая ветка
3. В узле графа содержащем вызов функции осуществляется интерпретация графа для вызываемой функции
4. В узле графа с конфигурацией осуществляется выполнение содержимых конструкций(контракций, рестрикций, подстановки конфигурации, ветвления)

Запуск и работа интерпретатора

Листинг 16: Компиляция текста программы интерпретатора на ограниченном Рефале

```
~/scp4/test$ srefc interpreter.ref ../reflib.ref
```

Листинг 17: Компиляция программы на языке Рефал-5 в промежуточное графовое представление

```
~/scp4/test$ inref4p program.ref
```

Листинг 18: Запуск исполняемого интерпретатора с указанием стратегии

```
~/scp4/test$ ./interpreter /lazy
```


4 Реализация интерпретатора

Интерпретирование в режиме запуска интерпретатора начинается со считывания файла с графовым представлением для функции «Go»

Листинг 19: Входная точка интерпретатора — функция Go

```
*$FROM ../reflib
$EXTERN Xin;

* $TRANSIENT Yes;
*$MATCHING ForRepeatedSpecialization;
*$STRATEGY Applicative;

/*
Go_6 returns graph which inref4 produced
CalledFunctions-Rec returns set of functions and theirs graph-representation
*/
$ENTRY Go {
    = <Prout <RunInterpreter <CalledFunctions-Rec (/ * scanned */ Go_6>>>;
}
```

Начало работы интерпретатора начинается с обхода файлов, порожденных компилятором исходной программы на Рефале-5 в промежуточное представление функции на языке входного Рефал-графа, которое является входным для суперкомпилятора SCP4. Для каждой функции ее представление находится в отдельном файле в папке «~/scp4/pgraph/».

Листинг 20: Обход и считывание для каждой функции ее графа

```
CalledFunctions -Rec {
  (e.Scanned) /* empty */ = e.Scanned;

  (e.Scanned) s.Next e.UnScanned =
    <CalledFunctions -Rec
      (e.Scanned s.Next)
      <MakeSet
        <SetDifference
          (e.UnScanned <CalledFunctions s.Next>)
          (e.Scanned s.Next)
        >
      >
    >;
}
```

Листинг 21: Функция возвращает список вызываемых из нее функций

```
CalledFunctions {
  s.FuncName =
    <MakeSet <NamesFromGraph s.FuncName <LoadGraph s.FuncName>>>;
}
```

Листинг 22: *Xxin* функция из библиотеки *reflib.ref* осуществляет считывание из файла

```
LoadGraph {
  s.FuncName = <Xxin '../pgraph/' <Explode s.FuncName>>;
}
```

Листинг 23: Извлечение из графов имен вызываемых функций

```
NamesFromGraph {
  s.FuncName (Fork e.Branches) = <NamesFromBranches e.Branches>;
  s.FuncName /* empty */ = <Prout 'Warning: no graph for ' s.FuncName>;
}
```

Осуществляется синтаксический анализ графового представления с целью извлечения информации.

Листинг 24: Извлечение из веток графов имен вызываемых функций

```
NamesFromBranches {  
  
  '+' (s.Num e.Branch) e.Branches =  
    <NamesFromBranch e.Branch>  
    <NamesFromBranches e.Branches>;  
  
  /* empty */ = /* empty */;  
}  
  
NamesFromBranch {  
  
  (Ct e.Contr) e.Branch = <NamesFromBranch e.Branch>;  
  
  (Fork e.Branches) e.Branch =  
    <NamesFromBranches e.Branches>  
    <NamesFromBranch e.Branch>;  
  
  (Conf e.Conf) e.Branch =  
    <NamesFromConf e.Conf>  
    <NamesFromBranch e.Branch>;  
  
  (Rs e.Restr) e.Branch = <NamesFromBranch e.Branch>;  
  
  (Def (e.Expr ':' t.Var)) e.Branch =  
    <NamesFromExpr e.Expr>  
    <NamesFromBranch e.Branch>;  
  
  /* empty */ = /* empty */;  
}
```

После получения нужных данных о программе запускается ее выполнение из функции *RunInterpreter*.

Листинг 25: Функция RunInterpreter

```
RunInterpreter {  
  
  e.Functions =  
    <Int  
      None  
      <Strategy>                                /*strategy mode*/  
      ('!' (Fn Go_6 (/*      */ ':' x)) ':' x)  
                                /*encode the task for supercompilation*/  
      (<LoadAllGraphs e.Functions>)  
    >;  
}
```

Листинг 26: Функция, формирующая структуру содержащую отображение названия функции в соответствующий ей граф

```
/*  
  <LoadAllGraphs s.FuncName*> == e.Program  
  
  e.Program ::= (s.FuncName e.Graph)*  
*/  
LoadAllGraphs {  
  
  s.FuncName e.Funcs =  
    (s.FuncName (<LoadGraph s.FuncName>))  
    <LoadAllGraphs e.Funcs>;  
  
  /* empty */ = /* empty */;  
}
```

Запуск и работа осуществляется в двух разных режимах — аппликативном и ленивом. Для этого необходимо задать флаги режимов в *.mst* файле или из командной строки, и считать.

Листинг 27: Обработка аргументов командной строки для установления стратегии вычислений интерпретатора

```
Strategy {  
  
  /* empty */  
  , <Upper <Arg 1>>  
  : {  
    '/APPL' = <Prout 'Applicative strategy'> Appl;  
    '/LAZY' = <Prout 'Lazy strategy'> Lazy;  
    e.Other = <Prout 'Applicative strategy (default)'> Appl;  
  }  
}
```

Функция *RunInterpreter* формирует вызов функции *Int* с подготовленными данными и установленными флагами. Два устанавливаемых флага для работы интерпретатора:

1. Флаг с указанием типа обрабатываемых веток — этот критерий влияет на подробность порождаемого дерева и в дальнейшем графа
2. Флаг с указанием стратегии вычисления

Листинг 28: Сигнатура функции Int

```
<Int s.ExtraBranches s.Mode e.Expr (e.Program)>  
s.ExtraBranches ::= None | ForSVars | All  
s.Mode ::= Lazy | Appl  
e.Expr ::= t.Term*  
t.Term ::=  
  ( '*' e.Expr )  
  | s.SYMBOL  
  | ( '!' (Fn s.Fn (e.Expr ':' t.Var)) ':' t.Var )
```

Функция *Int* осуществляет проверку на содержание в текущем поле зрения программы метакодированного вызова функции и метакодированного выражения в структурных скобках. Для разных случаев в зависимости от установ-

ленных режимов интерпретации запускаются разные программы дальнейшей интерпретации, а также передаются значения флагов.

Листинг 29: Реализация функции *Int*

```
Int {  
  
  s.ExtraBranches s.Mode ('*' e.Nested) e.Rest (e.Program) =  
    ('*' <Int s.ExtraBranches s.Mode e.Nested (e.Program)>)  
    <Int s.ExtraBranches s.Mode e.Rest (e.Program)>;  
  
  s.ExtraBranches Appl ('!' (Fn s.Fn (e.Arg ':' t.Var1)) ':' t.Var2)  
  e.Rest (e.Program) =  
    <IntFunc  
      s.ExtraBranches Appl s.Fn  
      <Int s.ExtraBranches Appl e.Arg (e.Program)>  
      (e.Program)  
    >  
    <Int s.ExtraBranches Appl e.Rest (e.Program)>;  
  
  s.ExtraBranches Lazy ('!' (Fn s.Fn (e.Arg ':' t.Var1)) ':' t.Var2)  
  e.Rest (e.Program) =  
    <Int  
      s.ExtraBranches Lazy  
      <IntFunc s.ExtraBranches Lazy s.Fn e.Arg (e.Program)>  
      (e.Program)  
    >  
    <Int s.ExtraBranches Lazy e.Rest (e.Program)>;  
  
  s.ExtraBranches s.Mode s.Symbol e.Rest (e.Program)  
  = s.Symbol <Int s.ExtraBranches s.Mode e.Rest (e.Program)>;  
  
  s.ExtraBranches s.Mode /* empty */ (e.Program) = /* empty */;  
}
```

Вызов функции интерпретируется в зависимости от режимов, но главное здесь — формирование нового вызова функции интерпретации графа, инициализация таблицы, стека возвратов, текста самого графа.

Листинг 30: Реализация интерпретации функции, встречающейся в графе

```
/*
  <IntFunc s.ExtraBranches s.Mode e.Expr (e.Program)> == e.ObjectExpr
*/
IntFunc {

  s.ExtraBranches Appl Residue s.FuncName e.Arg (e.Program)
    = <Call s.FuncName e.Arg>;

  s.ExtraBranches Appl Executable s.FuncName e.Arg (e.Program)
    = <Call s.FuncName e.Arg>;

  s.ExtraBranches Lazy Residue s.FuncName e.Arg (e.Program)
    = <Call s.FuncName <Int s.ExtraBranches Lazy e.Arg (e.Program)>>;

  s.ExtraBranches Lazy Executable s.FuncName e.Arg (e.Program)
    = <Call s.FuncName <Int s.ExtraBranches Lazy e.Arg (e.Program)>>;

  s.ExtraBranches s.Mode Appl___ e.Arg (e.Program)
    = <Appl___ <Int s.ExtraBranches s.Mode e.Arg (e.Program)>>;

  s.ExtraBranches s.Mode s.FuncName e.Arg (e.Program)
    = <IntGraph
      s.ExtraBranches
      s.Mode
      ((e.Arg ':' ('e' 1)))
      <LookupGraph s.FuncName e.Program>
      (() )
      (e.Program)
    >;
}
```

Далее будут приведены участок определения функции интерпретирующей граф и ее сигнатура.

Листинг 31: Сигнатура функции IntGraph

```
<IntGraph s.ExtraBranches s.Mode (e.Table) e.Graph (e.Backtrack) (e.Program)>
  == e.ObjectExpr

e.Table ::= (e.Value ':' (s.Type s.Index))*
e.Backtrack ::= (e.Table) e.Branches
e.Branches ::= e.Branch*
e.Branch ::= '+' (s.Number e.Graph)
```

Листинг 32: Участок определения функции IntGraph 1

```
IntGraph {

  s.ExtraBranches s.Mode (e.Table)
  (Ct e.Contr) e.Graph (e.Backtrack) (e.Program)
  , <PerformContr s.ExtraBranches s.Mode (e.Table) e.Contr (e.Program)>
  : {
    Ok e.NewTable
      = <IntGraph s.ExtraBranches s.Mode (e.NewTable) e.Graph
        (e.Backtrack) (e.Program)>;

    Fail = <IntGraph s.ExtraBranches s.Mode <Backtrack e.Backtrack>
      (e.Program)>;
  };

  s.ExtraBranches s.Mode (e.Table)
  (Fork '+' (s.Number e.Graph) e.Branches) (e.OldBacktrack)
  (e.Program)
  = <IntGraph s.ExtraBranches s.Mode (e.Table) e.Graph
    ((e.Table) e.Branches) (e.Program)>;

  s.ExtraBranches s.Mode (e.Table)
  (Rs e.Restr) e.Graph (e.Backtrack) (e.Program)
  , <PerformRestr (e.Table) e.Restr >
  : {
    Ok = <IntGraph s.ExtraBranches s.Mode (e.Table) e.Graph
      (e.Backtrack) (e.Program)>;
```


Листинг 33: Участок определения функции IntGraph 2

```

    Fail = <IntGraph s.ExtraBranches s.Mode <Backtrack e.Backtrack>
      (e.Program)>;
  };

  s.ExtraBranches Lazy (e.Table)
  (Conf (e.Expr ':' t.Var)) (e.Backtrack) (e.Program)
  = <Substitute (e.Table) e.Expr>;

  s.ExtraBranches Appl (e.Table)
  (Conf (e.Expr ':' t.Var)) (e.Backtrack) (e.Program)
  = <Int s.ExtraBranches Appl <Substitute (e.Table) e.Expr> (e.Program)>;

  s.ExtraBranches s.Mode (e.Table)
  (Def (e.Expr ':' (Var s.Type s.Index))) e.Graph (e.Backtrack) (e.Program)
  , <Int s.ExtraBranches s.Mode <Substitute (e.Table) e.Expr> (e.Program)>
  : e.Res, e.Table (e.Res ':' (s.Type s.Index)) : e.NewTable
  = <IntGraph s.ExtraBranches s.Mode (e.NewTable) e.Graph
    (e.Backtrack) (e.Program)>;
}

```

Рассмотрим сценарий интерпретации для конфигурации узла в графе с контракцией.

В функции *IntGraph* для такого случая осуществляется исполнение контракции, что подразумевает преобразования над таблицей. Дадим определение этой функции. Она будет возвращать в случае успеха/неуспеха ветку следующего исполнения по программе.

Листинг 34: Сигнатура функции исполняющей контракцию

```

/*
  <PerformContr (e.Table) e.Contr>
    == Ok e.Table
    == Fail
*/

```

Листинг 35: Определение функции исполняющей контракцию

```
PerformContr {  
  
  s.ExtraBranches s.Mode (e.Table) (Var s.Type s.Index) ':' e.Pattern (e.Program)  
  = <PerformContr-Aux  
    s.ExtraBranches  
    s.Mode  
    (e.Table)  
    (<LookupVar e.Table s.Type s.Index>)  
    (<SubstituteS (e.Table) e.Pattern>)  
    (e.Program)  
  >;  
}  
  
PerformContr-Aux {  
  
  s.ExtraBranches s.Mode (e.Table) (Success e.Value) (e.ConcretePattern)  
  (e.Program)  
  , <Match s.ExtraBranches s.Mode (e.Value) e.ConcretePattern (e.Program)>  
  : {  
    Ok e.NewVars = Ok <UpdateTable (e.Table) e.NewVars>;  
    Fail = Fail;  
  };  
}
```

На примере одного из узлов графа представления функции продемонстрирована ветка интерпретации. Аналогично интерпретация проходит для других узлов графа.

5 Тестирование

5.1 Исследование программ, где применение интерпретатора порождает более интересные остаточные программы

1. Суперкомпилируется интерпретатор int.ref с заданием int.mst :

Листинг 36: Задание на суперкомпиляцию интерпретатора по программе Trim

```
$DEFINE Prog
<Inref4
  Trim {
    ' ' e.X = <Trim e.X>;
    e.X ' ' = <Trim e.X>;
    e.X = e.X;
    /*empty*/ = /*empty*/;
  }
>;

<Int      e.X      (      ) > --
  <Trim    !    >    %.Prog    ;
```

int.ref (≈ 1000 строк) преобразуется в интерпретатор специализированный по программе с функцией Trim (≈ 40 строк) с оптимизированной операционной семантикой

Листинг 37: Программа, порожденная в результате суперкомпиляции интерпретатора по программе Trim 1

```
*$MST_FROM_ENTRY;

*$MATCHING ForRepeatedSpecialization ;
..
*$STRATEGY Applicative ;
```

Листинг 38: Программа, порожденная в результате суперкомпиляции интерпретатора по программе с функцией Trim 2

```
/*
$ENTRY Go {
  = <Prout <int1 e.X >> ;
}
*/
***

/*
*   InputFormat: <int1 e.1 >
*   OutputFormat: ==> e.0
*/
int1 {
  e.1 = <F74 e.1>;
}

/*
*   InputFormat: <F141 (e.1 ) s.144 >
*   OutputFormat: ==> s.191 (e.192 )
*/
F141 {
  (e.1 ' ') s.144, <F141 (e.1) s.144> : s.233 (e.234) = s.233 (e.234);
  (e.1) s.144 = s.144 (e.1);
}

/*
*   InputFormat: <F74 e.1 >
*   OutputFormat: ==> e.231
*/
F74 {
  ' ' e.1 = <F74 e.1>;
  s.144 e.1, <F141 (e.1) s.144> : s.191 (e.192) = s.191 e.192;
  e.1 ' ' = <F74 e.1>;
  e.1 = e.1;
}

***** The End *****
```

Если аргумент для функции Trim не начинается с пробела, но при этом заканчивается на 100 пробелов, всё равно 100 раз будет выполняться сопоставление с первым образцом. В остаточной программе такой избыточной проверки не будет.

2. Задание на суперкомпиляцию:

Листинг 39: Пример с заданием на суперкомпиляцию интерпретатора по программе верификации

```
\begin{lstlisting}
$DEFINE Prog
<Inref4
Pred {
  A B e.X = False;
  e.X = True;
}
Hello {
  A B e.X = Hello;
  e.X = e.X;
}
Go { e.X = <Pred <Hello e.X>>; }
>;

<Int All Appl      e.X      (      ) > --
      <Go      !      >      %.Prog      ;
```

Первая проекция Футамуры для интерпретатора, написанного в данной ВКР, и программы из задания на суперкомпиляцию получается следующая остаточная программа:

Листинг 40: Результат суперкомпиляции интерпретатора по программе верификации 1

```
*$MST_FROM_ENTRY;
*$MATCHING ForRepeatedSpecialization ;
*$STRATEGY Applicative ;
```

Листинг 41: Результат суперкомпиляции интерпретатора по программе верификации 2

```
/*
$ENTRY Go {
  = <Prout <int1 e.X >> ;
}
*/

***

/*
*   InputFormat: <int1 e.1 >
*   OutputFormat: ==> e.0
*/

int1 {
  e.1 = True;
}
```

Продemonстрированный пример порождает программу, которая верифицирует работу функции(описанной в задании на суперкомпиляцию), проверяющей получаемое слово на несодержание префикса «А В». Функция Hello никогда не возвращает строку с префиксом «А В». Значит работа функции Pred может быть оптимизирована до возврата единственного значения — True.

ЗАКЛЮЧЕНИЕ

В рамках ВКР реализован интерпретатор графа (порождаемого на вход суперкомпилятору SCP4). Интерпретатор сделан пригодным для суперкомпиляции и был суперкомпилирован. С разными функциями для интерпретации после работы суперкомпилятора над специализацией интерпретатора были получены тождественные исходным функциям функции, но не тождественные как алгоритмы, а оптимизированные. Эти примеры демонстрируют улучшение результатов суперкомпиляции SCP4 посредством применения промежуточного интерпретатора. А также дальнейшие возможности для поиска стратегий к улучшениям.

Также результатом ВКР является реализация ранее не использованной возможности MST-схем конструирования многоуровневых заданий на суперкомпиляцию.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А. П. Немытых, Суперкомпилятор SCP4. Общая структура, ЛКИ, 2007. — 152 стр., ISBN 9785382003658.
2. Турчин В.Ф. Феномен науки: Кибернетический подход к эволюции. Москва: Наука, 1993. 295 с.
3. Анд.В. Климов, Введение в метавычисления и суперкомпиляцию. В сб.: Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям. М.: Изд-во КомКнига, 2008. С. 343-368.
4. Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1986. — Vol. 8, no. 3. — Pp. 292–325.
5. Ключников И. Суперкомпиляция: идеи и методы // Практика функционального программирования. — 2011. — № 7. — С. 157-197.
6. Климов А.В., Романенко С.А. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М.В.Келдыша. 2018. No 111. 36 с. doi:10.20948/prepr-2018-111 URL: <http://library.keldysh.ru/preprint.asp?id=2018-111>
7. Futamura Y., Nogi K. Generalized partial computation // Of the IFIP TC2 Workshop. — Amsterdam: North-Holland Publishing Co., 1988, pp. 133–151
8. Турчин, В.Ф. РЕФАЛ-5 Руководство по программированию и справочник [Электронный ресурс] — Электронные данные. — Режим доступа: http://refal.ru/rf5_frm.htm
9. Анд. Климов, Введение в Рефал и суперкомпиляцию [Электронный ресурс] / Анд. Климов — Электронные данные. — Институт прикладной математики

им. М.В.Келдыша РАН, 2000. — Режим доступа:

<http://www.refal.ru/~klimov/lectures/>

10. Valentin F. Turchin, REFAL-5, Programming Guide and Reference Manual // The City College of New York, New England Publishing Co. Holyoke, 1989 — Режим доступа:<http://refal.botik.ru/book/html/>
11. V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In: Partial Evaluation, LNCS vol. 1110, 1996, pp. 481-509.

ПРИЛОЖЕНИЕ А

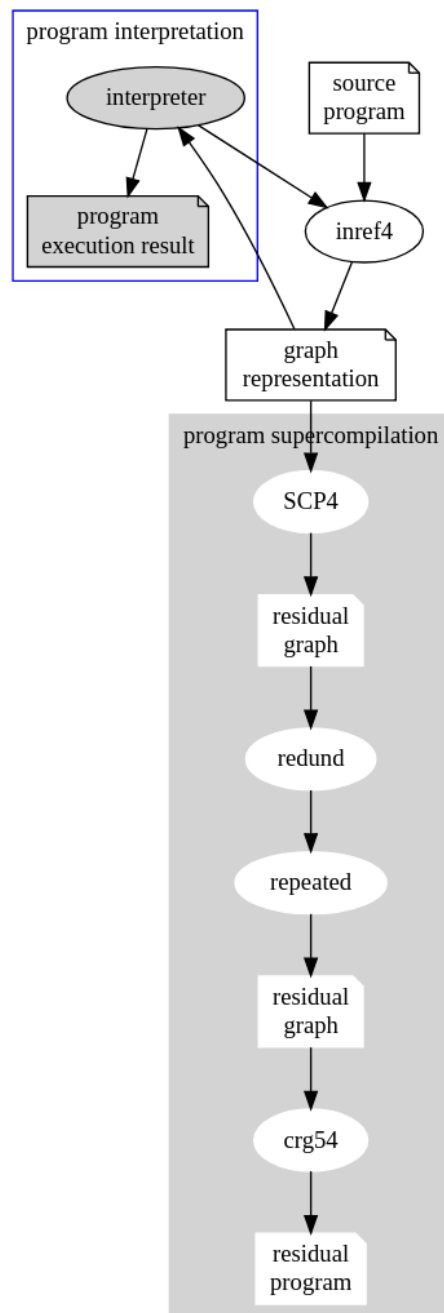


Рис. 7: Использование SCP4 и интерпретатора