Санкт-Петербургский государственный университет

Екатерина Петровна Винник

Выпускная квалификационная работа

Реализация дистиллятора для простого функционального языка на Haskell

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2018 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Системное программирование*

Научный руководитель:
к.ф.-м.н., доцент кафедры информатики, С.В. Григорьев

Консультант:
к.ф.-м.н., доцент кафедры системного программирования, Д.А. Березун

Рецензент:
программист ООО «Интеллиджей Лабс» Е.А. Вербицкая

Санкт-Петербург
2022

Saint Petersburg State University

Ekaterina Vinnik

Bachelor's Thesis

Implementation of Distiller for Model Language in Haskell

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2018 "Software and Administration of Information Systems"*

Profile: *Software Engineering*

Scientific supervisor:
C.Sc, docent. S.V. Grigoriev

Consultant:
C.Sc, docent. D.A. Berezun
Reviewer:
Software engineer at IntelliJ Labs Co. Ltd. E.A. Verbitskaia

Saint Petersburg
2022

Оглавление

Введение

Несмотря на мощность вычислительных устройств в настоящее время, операции обращения к памяти при исполнении программ до сих пор являются дорогостоящими. Обращения к памяти создаются за счет промежуточных структур данных, генерируемых во время исполнения программы, которые загружаются в память и выгружаются из нее. Трансформация программы таким образом, чтобы она во время исполнения использовала меньшее количество промежуточных структур, уменьшит количество обращений к памяти, а в идеальном случае сократит их количество до одного обращения, осуществляющего выгрузку результата программы. В качестве примера функции, исполнение которой повлечет создание промежуточной структуры данных, можно рассмотреть последовательность операций, написанную на неформальном языке, состоящую из двух применений функции map к массиву:

Array.map g (Array.map f arr)

Данный код естественнен для программиста при наличии библиотеки функций для работы с коллекциями, однако, без наличия оптимизаций, в нём будет создаваться промежуточный массив — результат применения функции f к массиву arr. Можно соединить исходную последовательность функций в одну таким образом, что вместо двух последовательных применений map к массиву arr будет применен один map с композицией исходных функций:

Array.map (g ∘ f) arr

Трансформированный код промежуточных структур данных не порождает и требует обращения к памяти только для записи результата. В общем случае не все последовательности функций, используемые для работы с коллекциями, могут быть соединены в одну очевидными преобразованиями. Для решения данной проблемы был разработан подход *Stream Fusion*. Библиотека *strymonas*, описанная в [13], реализующая этот подход, предоставляет набор простых операций, хорошо

подвергающихся соединению, с помощью которых можно составлять более сложные операции над коллекциями. Подобные решения достаточно популярны и реализованы для различных языков. Например, для Haskell[1] [2], Java [12], OCaml [13]. Недостатком этого подхода является то, что не все операции могут быть выражены через предоставляемый набор простых операций.

Проблема большого количества обращений к памяти, вызванных использованием промежуточных структур, актуальна не только для коллекций. Она свойственна для областей, в которых работа над данными осуществляется с помощью комбинирования набора простых операций: каждая из этих операций будет порождать результат, который будет передан в качестве входных данных следующей в последовательности операции. Такой подход используется в библиотеках для машинного обучения, например в *Tensorflow*, описанной в [16]. Проблема порождения промежуточных структур данных простыми функциями этой библиотеки также решается с помощью соединения этих функций, эту оптимизацию осуществляют инструменты *XLA* и *MLIR*, описанные в [1] и [11].

Работа с разреженными данными также часто осуществляется с помощью набора простых операций, последовательное исполнение которых приводит к созданию промежуточных структур данных во время исполнения. Такой подход реализовывают, например, библиотеки операций разреженной линейной алгебры. Они предоставляют набор примитивных операций для работы с разреженными данными (например, умножение двух матриц или их сложение), с помощью которых могут быть выражены более сложные операции.

Одним из перспективных направлений в данной области является стандарт GraphBLAS API[2] [4], описывающий набор примитивов разреженной линейной алгебры (матрицы, вектора) и операций над ними, необходимый для выражения алгоритмов анализа графов. Алгоритмы,

[1]Пакет Data.List.Stream поддерживает соединение операций для списков. Страница пакета на Hackage: https://hackage.haskell.org/package/stream-fusion-0.1.2.5/docs/Data-List-Stream.html.

[2]Спецификация GraphBLAS C API, версия 2.0.0: https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf

таким образом, являются последовательностью элементарных операций линейной алгебры, что и в данном случае приводит к проблеме промежуточных данных, которая для больших графов становится особенно критичной.

Операции над разреженными данными трудно поддаются соединению ввиду сложности используемых структур данных и алгоритмов их обработки, поэтому соединения комбинаций простых операций реализовываются вручную, как, например, в *GraphBlas:SuiteSparse* [3]. Классическим примером ручного объединения операций является операция muliplyAdd — объединение сложения и умножения матриц, или применение маски после бинарной операции, часто используемое в GraphBLAS для фильтрации результата операции. Недавнее исследование [19], проведённое при реализации GraphBLAS на GPGPU, показало, что современные подходы к оптимизациям программ не справляются с объединением операций разреженной линейной алгебры, а значит требуются более мощные подходы.

Stream Fusion как подход, позволяющий сократить использование промежуточных списков, часто излагается в контексте дефорестации — более мощного подхода, сокращающего использование промежуточных списков и деревьев, описанного в [18]. Суперкомпиляция [17], частичные вычисления [8] и дистилляция [5] также являются родственными дефорестации подходами, более мощными, чем *Stream fusion*. Поэтому в качестве подходов, которые потенциально могут быть использованы для оптимизации программ, содержащих операции разреженной линейной алгебры, естественно начать с исследования возможностей суперкомпиляции, частичных вычислений или дистилляции. Хотя суперкомпиляция, дефорестация и частичные вычисления достаточно распространены, дистилляция позволяет сократить использование промежуточных структур данных в большем множестве программ по сравнению с остальными существующими подходами согласно [5]. Другим преимуществом дистилляции является то, что применение этого подхода в некоторых случаях производит суперлинейное ускорение программ, которое не может быть достигнуто с помощью других подходов [5]. В ма-

гистерской работе [15] было показано, что суперкомпиляция, являющаяся более мощной, чем частичные вычисления и дефорестация, может достигать только линейного ускорения. Ввиду этих достоинств исследование возможностей дистилляции в качестве подхода к оптимизации программ представляет особенный интерес.

Хотя подход дистилляции позволяет трансформировать большее множество программ, чем остальные подходы, в силу своей трудоемкости он не так распространен, как суперкомпиляция, частичные вычисления и дефорестация. На данный момент существует единственная реализация подхода дистилляции — проект Distiller[3], но она не является стабильной. Поэтому не существует программы, демонстрирующей возможности дистилляции в качестве оптимизатора программ, использующих операции линейной алгебры. Реализация подхода дистилляции для базового функционального языка, обладающего хорошим тестовым покрытием и набором примеров работы подхода, помогла бы исследовать применимость подхода дистилляции для оптимизации программ, использующих операции линейной алгебры.

[3]Официальная документация и исходный код проекта могут быть найдены по ссылке https://github.com/poitin/Distiller. Дата последнего обращения 15.05.2022.

Постановка задачи

Целью данной работы является реализация упрощенного алгоритма дистилляции в существующем проекте Distiller[4] (прим. далее дистиллятор).

Для достижения этой цели были поставлены следующие задачи.

- Перепроектировать существующее решение, отделить интерфейс взаимодействия с пользователем от существующей реализации алгоритма дистилляции.
- Реализовать упрощённую версию алгоритма дистилляции на языке Haskell и интегрировать ее в проект Distiller (дистиллятор) на место предыдущей реализации. Haskell в качестве языка реализации выбран в силу того, что инфраструктура проекта Distiller создана на Haskell.
- Подготовить тестовую инфраструктуру для тестирования дистиллятора:
  - Выбрать и подключить тестовую платформу;
  - Подключить непрерывную интеграцию.
- Провести тестирование полученного дистиллятора с использованием различных методик.

[4]Официальная документация и исходный код проекта могут быть найдены по ссылке https://github.com/YaccConstructor/Distiller. Дата последнего обращения 15.05.2022.

1. Обзор

В данной главе будут представлены основные понятия и технологии, необходимые для дальнейшего описания работы:

- дистилляция в контексте суперкомпиляции и дефорестации;
- подход дистилляции;
- обзор существующих решений для тестирования реализованного алгоритма.

1.1. Метавычисления

Такие подходы к оптимизации программ, как дефорестация, суперкомпиляция и дистилляция, могут бороться с созданием промежуточных структур данных. Данные подходы часто обобщаются под понятием *метавычисления* и являются типичными для функциональных языков программирования, хотя имеются попытки применить их для объектно-ориентированных и императивных языков и порой относительно успешные [10], [9]. Базовые техники данных подходов как правило описываются (и реализовываются) для некоторого модельного языка с характерным для каждого подхода набором ограничений. Это, в частности, затрудняет экспериментальное сравнение перечисленных техник. Сами алгоритмы дефорестации, суперкомпиляции и дистилляции как правило описываются как наборы правил переписывания терма (программы). Фактически, это набор сопоставлений с образцом, по которому находится участок, к которому можно применить ту или иную трансформацию.

Подход дефорестации, хотя и является самым простым, требует от исходных функций выполнения условия безлесной формы [18] для того, чтобы быть оптимизированными. Требование этой формы является очень сильным ограничением, поэтому дефорестация является менее мощным подходом, чем суперкомпиляция. Это было формально показано в [14]. В качестве примера была рассмотрена программа, реали-

зующая алгоритм Кнутта-Мориса-Пратта, которая почти не оптимизируется с помощью дефорестации, но оптимизируется с помощью суперкомпиляции.

Так как дистилляция разработана на основе суперкомпиляции, она имеет процесс работы, аналогичный процессу работы алгоритма суперкомпиляции. Отличием дистилляции от суперкомпиляции является то, что она описывает более широкий набор сопоставлений с образцом, чем суперкомпиляция. Дистилляция является обобщением суперкомпиляции и позволяет трансформировать большее количество программ [5]. Более того, на основе дистилляции можно построить невырожденную иерархию преобразователей программ, на самом нижнем уровне которой находится суперкомпилятор, далее дистиллятор, за которым идут преобразователи более высоких уровней [7]. Поэтому подход суперкомпиляции отдельно рассматриваться не будет.

1.2. Подход дистилляции

В данном разделе будут введены основные определения и теоремы, используемые в данной работе, которые можно найти в статье [6]. Подход дистилляции будет изложен для языка, приведенного в 1, описанного в статье [6].

$$
\begin{array}{lll}
e ::= x & & \text{Variable} \\
\quad \mid c\ e_1 \ldots e_k & & \text{Constructor Application} \\
\quad \mid \lambda x.e & & \lambda\text{-Abstraction} \\
\quad \mid f & & \text{Function Call} \\
\quad \mid e_0\ e_1 & & \text{Application} \\
\quad \mid \textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k & & \text{Case Expression} \\
\quad \mid \textbf{let}\ x = e_0\ \textbf{in}\ e_1 & & \text{Let Expression} \\
\quad \mid e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n & & \text{Local Function Definitions} \\
\\
p ::= c\ x_1 \ldots x_k & & \text{Pattern}
\end{array}
\tag{1}
$$

Язык обладает стандартной операционной семантикой со стратегией

вычисления *вызов-по-имени* (*call-by-name*).

Программа, написанная на языке из формулы 1 — это выражение, являющееся переменной, вызовом конструктора, операцией применения (являющейся левоассоциативной в этом языке), λ-абстракцией, вызовом функции, оператором *case*, *let* или *where*. Так как функциональный вызов $f\ x_1\ \ldots x_n$ может быть описан с помощью последовательности операций применения функции f к своим аргументам $((f\ x_1)\ \ldots)\ x_n$, хранить функцию вместе с аргументами не требуется. Поэтому для обозначения функционального вызова используется только имя функции f без аргументов. Переменные, введенные в λ-абстракции, *let* или *case*, считаются связанными, все остальные переменные считаются свободным. Предполагается, что $e_1 \equiv e_2$, если e_1 одинаковы e_2 с точностью до переименования связанных переменных, т.е., если e_1 и e_2 α-эквивалентны. Также предполагается, что входная программа не содержит *let* выражений и они могут быть получены только в процессе трансформации программы. Каждый конструктор имеет фиксированную арность — к примеру, конструктор пустого списка Nil имеет арность 0, а для выражения $c\ e_1 \ldots e_n$ арность конструктора c обязана быть равной n. Также предполагается, что ветки выражения *case* не перекрываются и покрывают все случаи.

На рисунке 1 приведена программа, демонстрирующая синтаксис введенного языка, которая вычисляет n-ое число Фибоначчи. Натуральные числа заданы с помощью арифметики Пеано: нуль определяется с помощью конструктора нулевой арности Z, все остальные числа задаются с помощью конструктора с одним аргументом $S\ n$, где n — число Пеано. Программа является выражением вида $e_0\ where\ f_1 = e_1 \ldots f_2 = e_2$, где e_0 — это функциональный вызов $fib\ n$, $f_1 = e_1$ из определения языка 1 — определение функции fib, $f_2 = e_2$ из определения языка 1 — определение функции *add*. Оператор *case* для fib использует связанную λ-абстракцией переменную n. Выражение $Z \Rightarrow S\ Z$ соответствует первой ветке оператора *case* из определения языка — $p_1 \Rightarrow e_1$, где e_1 имеет вид $c\ e_1$, так как S является конструктором арности один с аргументом n. $S\ n' \Rightarrow case\ n'\ of$ ... соответствует второй ветке оператора

$case\ -\ p_2 \Rightarrow e_2$. Выражение $add\ (fib\ n'')\ (fib\ n')$ во вложенном $case$ соответствует выражению $e_0\ e_1$ из определения языка, то есть, означает левоассоциативную операцию применения $add\ (fib\ n'')$ к $(fib\ n')$. Оператор $case$ для функции add интерпретируется аналогично.

```
fib n
where
fib  = λn. case n of
                 Z    ⇒ S Z
               | S n' ⇒ case n' of
                            Z    ⇒ S Z
                          | S n'' ⇒ add (fib n'') (fib n')
add = λx.λy. case x of
                 Z    ⇒ y
               | S x' ⇒ S (add x' y)
```

Рис. 1: Программа, вычисляющая n-ое число Фибоначчи

В качестве еще одного примера можно рассмотреть программу, запускающую функцию $qrev$ на списках xs и $[]$, приведенную на рисунке 2. Функция $qrev$, работающая с двумя списками xs и ys, разворачивает список xs и присоединяет к ys.

```
qrev xs []
where
qrev xs ys = case xs of
                 Nil          ⇒ ys
               | Cons(x, xs) ⇒ qrev xs Cons(x, ys)
```

Рис. 2: Программа, конкатенирующая перевернутый список xs и ys

Определение 1.2.1 *Подстановкой* называется отображение $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$, сопоставляющее переменной x_i из набора $x_1 \ldots x_n$ выражение e_i из набора $e_1 \ldots e_n$, которое должно быть подставлено вместо x_i. Результат подстановки выражений $e_1, \ldots, e_n$, соответствующих $x_1, \ldots, x_n$, в выражение e с учетом переименования свободных переменных так, чтобы их названия не пересекались с множеством названий связных переменных, обозначается $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$.

Определение 1.2.2 *Переименованием* называется биективное отображение $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$. Выражение $e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ — результат замены переменных $x_1, \ldots, x_n$ соответствующих переменным $x'_1, \ldots, x'_n$ для выражения e.

Определение 1.2.3 Контекст редукции $\mathcal{R}$ — выражение, в которое можно подставить какое-либо выражение на место редекса, помеченного знаком $\bullet$, которое может иметь следующий вид:

$$\mathcal{R} ::= \bullet\ e \mid (\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ \ldots\ \mid p_k \Rightarrow e_k)$$

Определение 1.2.4 Контекст вычисления $\mathcal{E}$ представляется как последовательность вложенных друг в друга контекстов редукции, каждый из которых имеет следующий вид.

$$\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{R} : \mathcal{E} \rangle$$

Определение 1.2.5 Вставка выражения e в вычислительный контекст κ, обозначаемая как $\kappa\bullet e$, определена в формуле 1.2.5.

$$\begin{aligned}
\langle\rangle\bullet e &= e \\
\langle(\bullet\ e') : \kappa\rangle\bullet e &= \kappa\bullet(e\ e') \\
\langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ \ldots\ \mid p_k \Rightarrow e_k) : \kappa\rangle\bullet e & \\
&= \kappa\bullet(\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ \ldots\ \mid p_k \Rightarrow e_k)
\end{aligned}$$

Определение 1.2.6 Подстановка определения функции (*unfolding*) в редекс выражения e с функциональным окружением Δ делается по правилам, определенным в формуле 2. В параметре k, фигурирующем в приведенных правилах, хранится контекст редекса, в параметре ρ хранятся пары переменных и соответствующих им значений $x_i \to e_i$, кото-

рые содержатся в выражениях вида $let\ x_i\ =\ e_i\ in\ e'_i$.

$$\mathcal{U}[\![e]\!]\ \Delta = \mathcal{U}'[\![e]\!]\ \langle\rangle\ \varnothing\ \Delta$$

$$\begin{array}{lcl}
\mathcal{U}'[\![x]\!]\ \kappa\ \rho\ \Delta & = & \begin{cases} \mathcal{U}'[\![\rho(x)]\!]\ \kappa\ \rho\ \Delta, & \text{if } x \in dom(\rho) \\ \kappa\bullet x, & \text{otherwise} \end{cases} \\
\mathcal{U}'[\![c\ e_1 \ldots e_k]\!]\ \kappa\ \rho\ \Delta & = & \kappa\bullet(c\ e_1 \ldots e_k) \\
\mathcal{U}'[\![\lambda x.e]\!]\ \kappa\ \rho\ \Delta & = & \kappa\bullet(\lambda x.e) \\
\mathcal{U}'[\![f]\!]\ \kappa\ \rho\ \Delta & = & \kappa\bullet e \text{ where } (f = e) \in \Delta \\
\mathcal{U}'[\![e_0\ e_1]\!]\ \kappa\ \rho\ \Delta & = & \mathcal{U}'[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \Delta \\
\mathcal{U}'[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \Delta & & \\
\qquad = \ \mathcal{U}'[\![e_0]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \Delta & & \\
\mathcal{U}'[\![\textbf{let}\ x = e_0\ \textbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta & = & \textbf{let}\ x = e_0\ \textbf{in}\ \mathcal{U}'[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \Delta \\
\mathcal{U}'[\![e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \Delta & & \\
\qquad = \ \mathcal{U}'[\![e_0]\!]\ \kappa\ \rho\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) & & \\
\qquad\quad \textbf{where}\ f_1 = e_1 \ldots f_n = e_n & &
\end{array} \tag{2}$$

Определение 1.2.7 Помеченная система переходов (*labelled transition system*), построенная по программе, являющейся выражением e, определяется как четырехместный кортеж $t = (\mathcal{E}, e, \rightarrow, Act)$ где:

- $\mathcal{E}$ — набор *состояний* помеченной системы переходов, каждое из которых является либо выражением, либо конечным состоянием **0**;
- t содержит в качестве корня выражение e, то есть, $root(t) = e$;
- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ — *отношение перехода*, связывающее два состояния с помощью операций, приведенных в формуле 3. Запись $e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)$ для помеченной системы переходов с корневым состоянием e означает, что помеченные системы переходов $t_1 \ldots t_n$ были получены из e в результате переходов, имеющих метки $\alpha_1 \ldots \alpha_n$;
- если выражение $e \in \mathcal{E}$ и $(e, \alpha, e') \in \rightarrow$, то $e' \in \mathcal{E}$;
- Act — набор меток α, отмечающих совершаемые переходы, определенных следующим образом: x — переменная, c — метка конструктора, @ — метка применения функции, $\#i$ — метка i аргумента, λx — абстракция над переменной x, **case** — оператор *case*,

p — ветка оператора $case$, **let** — оператор, τ_f — подстановка определения функции f, τ_c — подстановка конструктора c, или τ_β — β-редукция.

$$
\begin{array}{ll}
\mathcal{L}[\![x]\!]\ \rho\ \Delta & = x \to (x, \mathbf{0}) \\
\mathcal{L}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \Delta & = (c\ e_1 \ldots e_n) \to (c, \mathbf{0}), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (\#n, \mathcal{L}[\![e_n]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![\lambda x.e]\!]\ \rho\ \Delta & = (\lambda x.e) \to (\lambda x, \mathcal{L}[\![e]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![f]\!]\ \rho\ \Delta & = \begin{cases} f \to (\tau_f, \mathbf{0}), & \text{if } \ f \in \rho \\ f \to (\tau_f, \mathcal{L}[\![e]\!]\ (\rho \cup \{f\})\ \Delta), & \text{otherwise} \quad \text{where } \ (f = e) \in \Delta \end{cases} \\
\mathcal{L}[\![e_0\ e_1]\!]\ \rho\ \Delta & = (e_0\ e_1) \to (@, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![e = (\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k)]\!]\ \rho\ \Delta & \\
\qquad = e \to (\textbf{case}, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (p_1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (p_k, \mathcal{L}[\![e_k]\!]\ \rho\ \Delta) & \\
\mathcal{L}[\![e = (\textbf{let}\ x = e_0\ \textbf{in}\ e_1)]\!]\ \rho\ \Delta & \\
\qquad = e \to (\textbf{let}, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), (x, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta) &
\end{array}
\tag{3}
$$

Параметр ρ, фигурирующий в правилах 3, используется для накопления функциональных вызовов — он хранит названия вызванных функций, параметр Δ хранит набор определений функций, т.е., является функциональным окружением. Приведенные в 3 правила применяются для входного выражения e сверху вниз путем сопоставления с образцом. Например, в случае, если выражение e является переменной вида x, будет применено первое сверху правило, в случае, если выражение e является конструктором с n аргументами, будет применено второе сверху правило.

В качестве примеров помеченных систем переходов, построенных по программе с использованием правил 3, можно рассмотреть системы переходов 3, 4, построенные по программам 2 и 1 соответственно.

Определение 1.2.8 Помеченная система переходов t_1 состоит в отношении переименования с помеченной системой переходов t_2 тогда и только тогда, когда существует переименование σ такое, что $t_1 \backsimeq_\sigma^\varnothing t_2$, где рефлексивное, транзитивное и симметричное отношение $\backsimeq_\sigma^\rho$ определено в формуле 4.

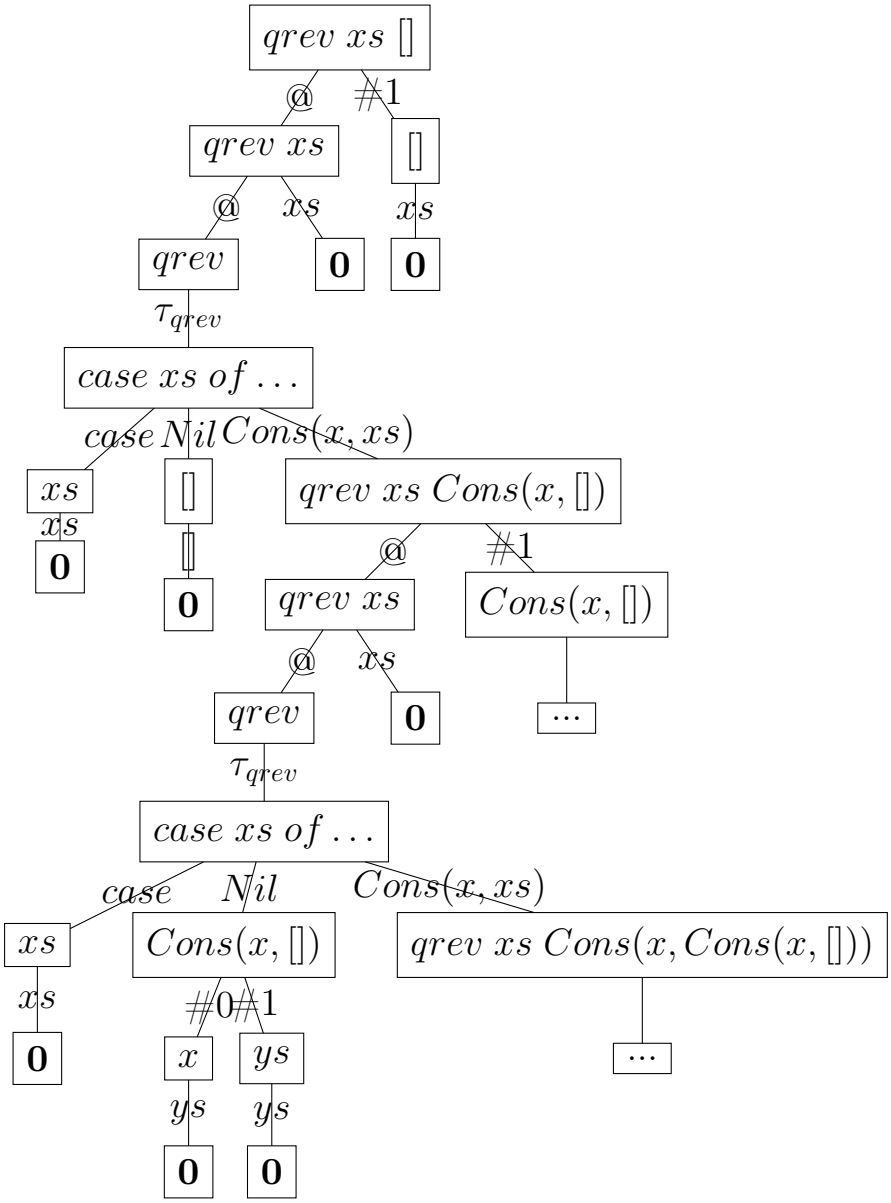

Рис. 3: Помеченная система переходов, построенная по программе 2

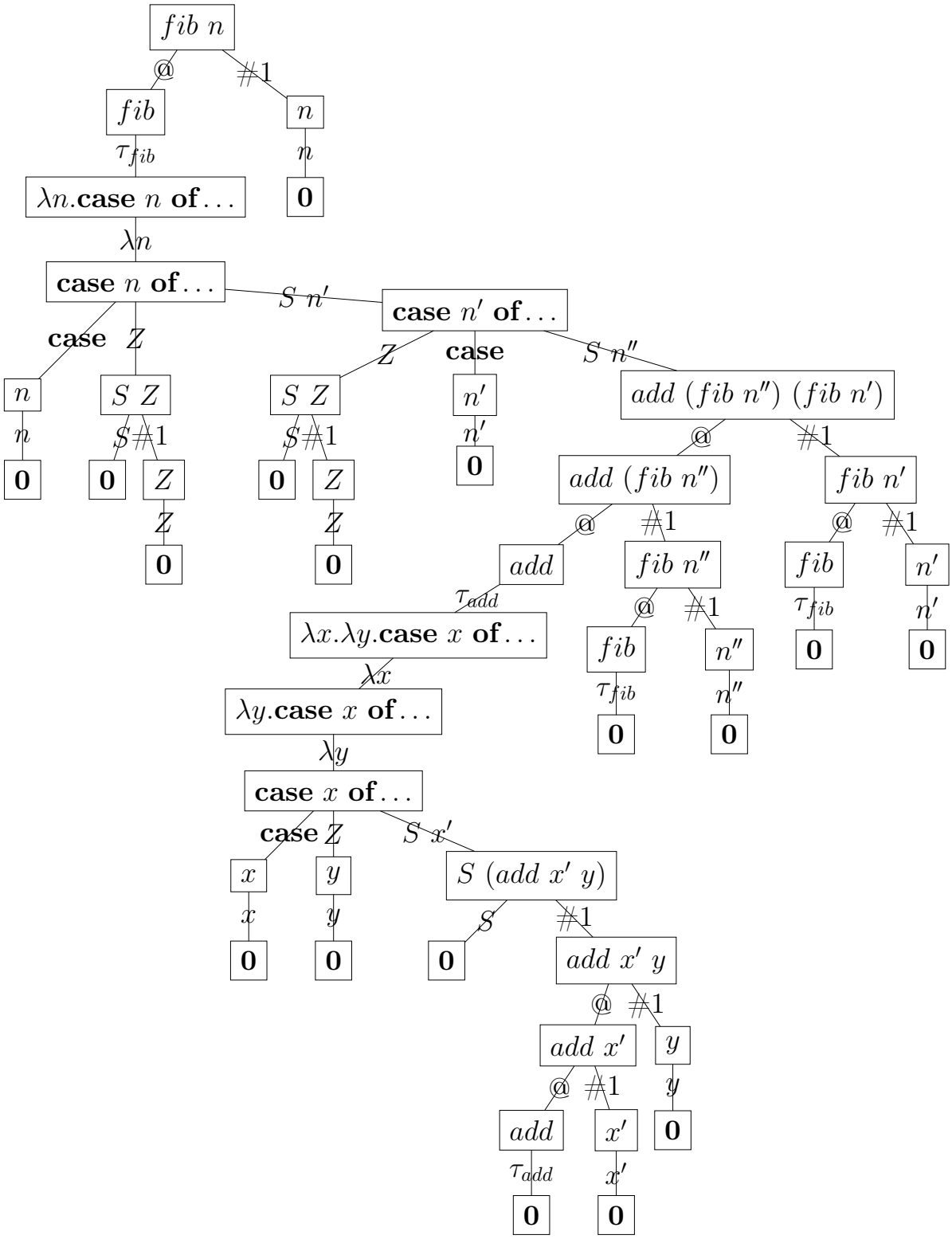

Рис. 4: Помеченная система переходов, построенная по программе 1

$$
\begin{array}{ll}
(x \to (x, \mathbf{0})) \approx^{\rho}_{\sigma} (x' \to (x', \mathbf{0})), & \text{if } x\sigma = x' \\
(e \to (\tau_f, t)) \approx^{\rho}_{\sigma} (e' \to (\tau_{f'}, t')), & \text{if } ((f, f') \in \rho) \vee (t \approx^{\rho \cup \{(f,f')\}}_{\sigma} t') \\
(e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \approx^{\rho}_{\sigma} (e' \to (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), & \\
\qquad \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \approx^{\rho}_{\sigma \cup \sigma'} t'_i))
\end{array}
\tag{4}
$$

Определение 1.2.9 Помеченная система переходов t_1 *гомеоморфно вложена* в помеченную систему переходов t_2 в том и только в том случае, если существует переименование σ, такое, что $t_1 \lesssim^{\varnothing}_{\sigma} t_2$, где рефлексивное, транзитивное и антисимметричное отношение $\lesssim^{\rho}_{\sigma}$ определено в формуле 5.

$$
\begin{array}{ll}
t \lesssim^{\rho}_{\sigma} t', & \text{if } (t \lhd^{\rho}_{\sigma} t') \vee (t \bowtie^{\rho}_{\sigma} t') \\
t \lhd^{\rho}_{\sigma} (e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)), & \text{if } \exists i \in \{1 \ldots n\} \bullet t \lesssim^{\rho}_{\sigma} t_i \\
(x \to (x, \mathbf{0})) \bowtie^{\rho}_{\sigma} (x' \to (x', \mathbf{0})), & \text{if } x\sigma = x' \\
(e \to (\tau_f, t)) \bowtie^{\rho}_{\sigma} (e' \to (\tau_{f'}, t')), & \text{if } ((f, f') \in \rho) \vee (t \lesssim^{\rho \cup \{(f,f')\}}_{\sigma} t') \\
(e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \bowtie^{\rho}_{\sigma} (e' \to (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), & \\
\qquad \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \lesssim^{\rho}_{\sigma \cup \sigma'} t'_i))
\end{array}
\tag{5}
$$

Гомеоморфное вложение двух помеченных систем переходов является индикатором того, что рассматриваемая на данный момент программа может быть оптимизирована.

В качестве примера гомеоморфно вложенных систем переходов можно рассмотреть системы, приведенные на рисунке 5, где система справа является гомеоморфно вложенной в систему слева. Отношение гомеоморфного вложения выполняется, так как помеченная система переходов справа в точности повторяет систему переходов слева, за исключением того, что в качестве списка ys в ней используется пустой список. Так как пустой список входит в множество значений, принимаемое списком ys, система является гомеоморфно вложенной.

Определение 1.2.10 Программа может быть построена по помеченной системе переходов по правилам, определенным в формуле 6.

Параметр ε хранит набор функциональных вызовов, которые были сделаны, и ассоциирует их с вызовами нововведенных функций.

Эти правила также именуются правилами построения остаточной программы (*residualization rules*). В процессе работы правил построения остаточной программы могут быть введены новые функции, вызов

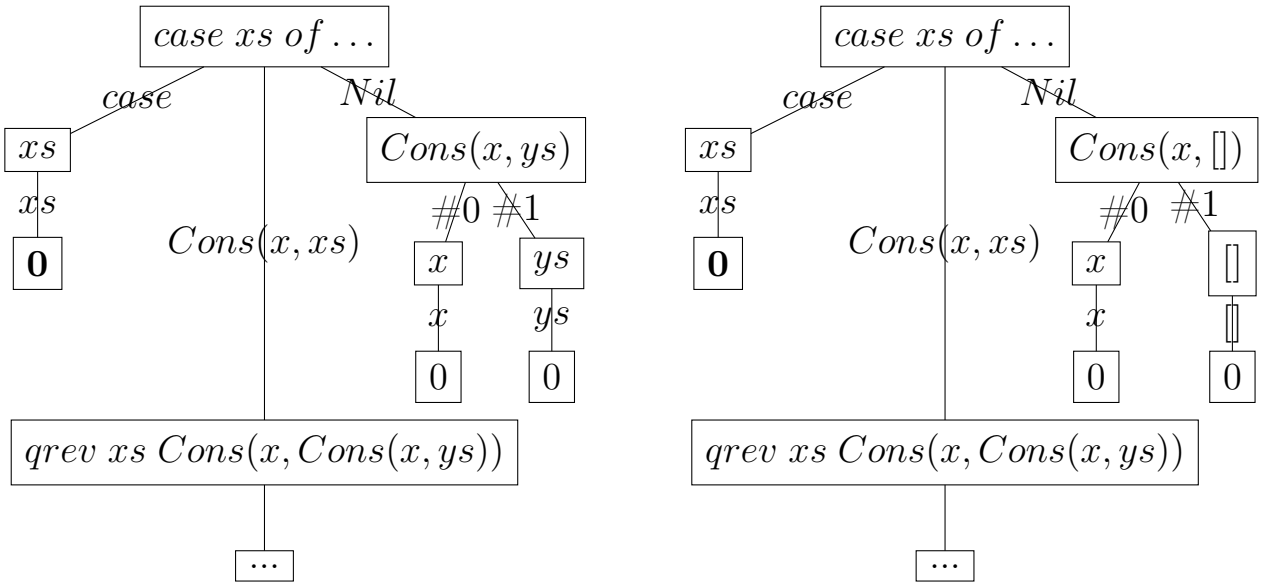

Рис. 5: Примеры вложенных помеченных систем переходов

которых обозначает функциональные вызовы, которые ранее встречались во время построения остаточной программы. Это позволяет оптимизировать процесс построения остаточной программы — в случае, если в помеченной системе переходов встретилось несколько одинаковых с точностью до переименования функциональных вызовов, помеченные системы переходов, соответствующие им, будут одинаковы, и соответственно, будут одинаковы остаточные программы.

Оптимизация программы в процессе алгоритма дистилляции заключается в модификации обрабатываемой помеченной системы переходов с использованием какой-либо из построенных ранее в алгоритме помеченных систем переходов и состоит из двух этапов. Первый этап, именуемый *срабатыванием свистка*, заключается в выполнении отношения гомеоморфного вложения для обрабатываемой помеченной системы переходов и построенной ранее помеченной системы переходов. Второй этап заключается в применении функции модификации, которая именуется обобщающей функцией и определена в 1.2.11, к обрабатываемой помеченной системе переходов с использованием построенной ранее помеченной системы переходов. Эта функция позволяет сократить длины путей от корневой вершины до конечных состояний в помеченной системе переходов. Это в свою очередь позволяет сократить количество

функциональных вызовов во время работы программы, а значит, сократить и количество обращений к памяти.

$$\mathcal{R}[\![t]\!] = \mathcal{R}'[\![t]\!]\ \varnothing \tag{6}$$

$$\begin{aligned}
&\mathcal{R}'[\![e \to (x, \mathbf{0})]\!]\ \varepsilon\ = x \\
&\mathcal{R}'[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon \\
&\qquad\qquad = c\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
&\mathcal{R}'[\![e \to (\lambda x, t)]\!]\ \varepsilon = \lambda x.(\mathcal{R}'[\![t]\!]\ \varepsilon) \\
&\mathcal{R}'[\![e \to (@, t_0), (\#1, t_1)]\!]\ \varepsilon \\
&\qquad\qquad = (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \\
&\mathcal{R}'[\![e \to (\mathbf{case}, t_0)(p_1, t_1), \ldots, (p_n, t_k)]\!]\ \varepsilon \\
&\qquad\qquad = \mathbf{case}\ (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\ |\cdots|\ p_k \Rightarrow (\mathcal{R}'[\![t_k]\!]\ \varepsilon) \\
&\mathcal{R}'[\![e \to (\mathbf{let}, t_0), (x_1, t_1), \ldots, (x_n, t_n)]\!]\ \varepsilon \\
&\qquad\qquad = \mathbf{let}\ x_1 = (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\ \mathbf{in} \ldots \mathbf{let}\ x_n = (\mathcal{R}'[\![t_n]\!]\ \varepsilon)\ \mathbf{in}\ (\mathcal{R}'[\![t_0]\!]\ \varepsilon) \\
&\mathcal{R}'[\![e \to (\tau_f, t)]\!]\ \varepsilon\ = \begin{cases} e'\theta, if \exists (e' = e'') \in \varepsilon \bullet e \equiv e''\theta \\ f'\ x_1 \ldots x_n \mathbf{where} f' = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \quad \text{otherwise}\ \ (f'\ \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases} \\
&\mathcal{R}'[\![e \to (\tau_\beta, t)]\!]\ \varepsilon = \mathcal{R}'[\![t]\!]\ \varepsilon \\
&\mathcal{R}'[\![e \to (\tau_c, t)]\!]\ \varepsilon = \mathcal{R}'[\![t]\!]\ \varepsilon
\end{aligned}$$

Определение 1.2.11 Функция $\mathcal{G}[\![t]\!][\![t']\!]\ \theta$, обобщающая помеченную систему переходов t с использованием t', определена в формуле 7, где θ — набор предыдущих обобщений, которые могут быть переиспользованы.

Правила обобщения (*generalization rules*) вызываются, когда в процессе работы алгоритма находится помеченная система переходов t, состоящая в отношении гомеоморфного вложения с системой t', встречавшейся ранее. Результатом функции обобщения является модификация системы t, в которой некоторые части были извлечены в отдельные с помощью **let**. В качестве примера можно рассмотреть результат обобщения систем, представленных на рисунке 5. Результат обобщения представлен на рисунке 6, обобщение произведено для системы переходов, использующей пустой список в качестве ys. В переходе с меткой **let** содержится значение ys (пустой список), которое будет подставлено в качестве ys после вычисления перехода с меткой τ_{qrev}, содержащего систему переходов для произвольного ys.

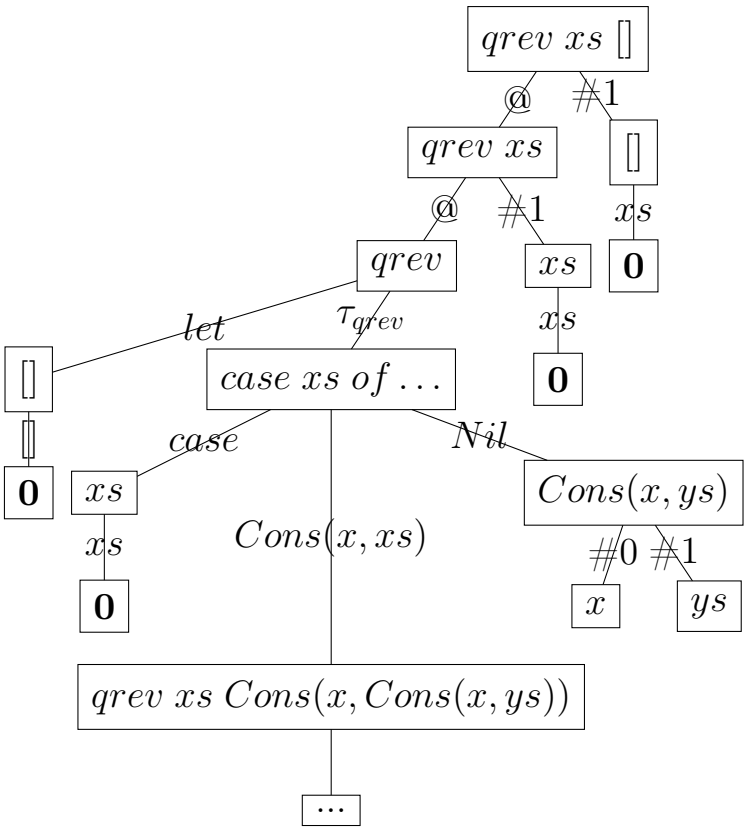

Рис. 6: Результат обобщения

$\mathcal{G}[\![t]\!][\![t']\!]\ \theta = \mathcal{A}[\![t^g]\!]\ \theta'$
where
$(t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \varnothing\ \varnothing$

$$
\begin{array}{lcl}
\mathcal{G}'[\![e \to (x, \mathbf{0})]\!][\![e' \to (x', \mathbf{0})]\!]\ \theta\ \gamma\ \rho & = & ((e \to (x, \mathbf{0})), \varnothing) \\
\mathcal{G}'[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!][\![e' \to (c, \mathbf{0}), (\#1, t'_1), \ldots, (\#n, t'_n)]\!]\ \theta\ \gamma\ \rho & & \\
 & = & ((e \to (c, \mathbf{0}), (\#1, t_1^g), \ldots, (\#n, t_n^g)), \bigcup_{i=1}^{n} \theta_i) \\
 & & \text{where} \\
 & & \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![e \to (\lambda x, t)]\!][\![e' \to (\lambda x', t')]\!]\ \theta\ \gamma\ \rho & = & ((e \to (\lambda x, t^g)), \theta') \\
 & & \text{where} \\
 & & (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ (\gamma \cup \{x\})\ \rho \\
\mathcal{G}'[\![e \to (@, t_0), (\#1, t_1)]\!][\![e' \to (@, t'_0), (\#1, t'_1)]\!]\ \theta\ \gamma\ \rho & & \\
 & = & ((e \to (@, t_0^g), (\#1, t_1^g)), \theta_0 \cup \theta_1) \\
 & & \text{where} \\
 & & \forall i \in \{0, 1\} \bullet (t_i^g, \theta_i) = \mathcal{G}[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![e \to (\mathbf{case}, t_0), (p_1, t_1), \ldots, (p_n, t_n)]\!][\![e' \to (\mathbf{case}, t'_0), (p'_1, t'_1), \ldots, (p'_n, t'_n)]\!]\ \theta\ \gamma\ \rho & & \\
 & = & ((e \to (\mathbf{case}, t_0^g), (p_1, t_1^g), \ldots, (p_n, t_n^g)), \bigcup_{i=0}^{n} \theta_i) \\
 & & \text{where} \\
 & & \forall i \in \{1 \ldots n\} \bullet (\exists \sigma \bullet p_i \equiv p'_i \sigma) \\
 & & (t_0^g, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t'_0]\!]\ \theta\ \gamma\ \rho \\
 & & \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ (\gamma \cup fv(p_i))\ \rho \\
\mathcal{G}'[\![e \to (\mathbf{let}, t_0), (x_1, t_1), \ldots, (x_n, t_n)]\!][\![e' \to (\mathbf{let}, t'_0), (x'_1, t'_1), \ldots, (x'_n, t'_n)]\!]\ \theta\ \gamma\ \rho & & \\
 & = & ((e \to (\mathbf{let}, t_0^g), (x_1, t_1^g), \ldots, (x_n, t_n^g)), \bigcup_{i=0}^{n} \theta_i) \\
 & & \text{where} \\
 & & (t_0^g, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t'_0]\!]\ \theta\ \gamma\ \rho \\
 & & \forall i \in \{1 \ldots n\} \bullet (t_i^g, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![e \to (\tau_f, t)]\!][\![e' \to (\tau_{f'}, t')]\!]\ \theta\ \gamma\ \rho & = & \left\{ \begin{array}{l} ((e \to (\tau_f, t)), \varnothing), \quad if\ (f, f') \in \rho \\ ((e \to (\tau_f, t^g)), \theta'), \quad \text{otherwise} \\ where \\ (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ (\rho \cup \{(f, f')\}) \end{array} \right. \\
\mathcal{G}'[\![e \to (\tau_\beta, t)]\!][\![e' \to (\tau_\beta, t')]\!]\ \theta\ \gamma\ \rho & = & \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![e \to (\tau_c, t)]\!][\![e' \to (\tau_c, t')]\!]\ \theta\ \gamma\ \rho & = & \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho & = & \left\{ \begin{array}{ll} (\mathcal{B}[\![x \to (x, \mathbf{0})]\!]\ \gamma', \varnothing), & if \exists (x, t_1) \in \theta \bullet t_1 \approx_{\varnothing}^{\varnothing} t_2 \\ (\mathcal{B}[\![x \to (x, \mathbf{0})]\!]\ \gamma', \{x \mapsto t_2\}), & \text{otherwise} \quad (\text{x is fresh}) \end{array} \right. \\
 & & \text{where} \\
 & & \gamma' = fv(t) \cap \gamma \\
 & & t_2 = \mathcal{C}[\![t]\!]\ \gamma'
\end{array}
\tag{7}
$$

$\mathcal{A}[\![t]\!]\ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} = root(t) \to (\mathbf{let}, t), (x_1, t_1), \ldots, (x_n, t_n)$

$\mathcal{B}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \to (@, (\ldots root(t) \to (@, t), (\#1, x_1 \to (x_1, \mathbf{0})) \ldots)), (\#1, x_n \to (x_n, \mathbf{0}))$

$\mathcal{C}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \to (\lambda x_1, \ldots root(t) \to (\lambda x_n, t) \ldots)$

Все описанные ранее правила предназначены для выполнения опре-

деленной задачи алгоритма — построения остаточной программы, построения помеченной системы переходов, обобщения систем переходов и других. Выполнение этих правил вызывается правилами трансформации разного уровня, приведенными в 8 и 9. Нулевой уровень трансформации отличается от остальных и приводится отдельно.

Определение 1.2.12 Правила трансформации уровня 0 определены в формуле 8.

$$
\begin{array}{l}
\mathcal{T}_0[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta \quad = \mathcal{T}_0'[\![x \to (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (c, \mathbf{0}), (\#1, \mathcal{T}_0[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_0[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_c, \mathcal{T}_0[\![e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (\lambda x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_\beta, \mathcal{T}_0[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta \quad = \begin{cases} (\kappa \bullet f) \to (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx_\sigma^\varnothing t \\ \mathcal{T}_0[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie_\sigma^\varnothing t \\ (\kappa \bullet f) \to (\tau_f, \mathcal{T}_0[\![\mathcal{U}[\![\kappa \bullet f]\!]\ \boxtimes]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \Delta), & \text{otherwise} \end{cases} \\
\qquad\quad \text{where } t = \mathcal{L}[\![\kappa \bullet f]\!]\ \varnothing\ \Delta \\
\mathcal{T}_0[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{T}_0[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_0[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\mathbf{let}, \mathcal{T}_0[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_0[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
\tag{8}
$$

$$
\begin{array}{l}
\mathcal{T}_0'[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \quad = t \\
\mathcal{T}_0'[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_0'[\![(t\ e) \to (@, t), (\#1, \mathcal{T}_0[\![e]\!]\ \langle\rangle\ \rho\ \theta\ \Delta)]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_0'[\![x \to (x, \mathbf{0})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') \to \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![(\kappa \bullet e_1')\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_0[\![(\kappa \bullet e_k')\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_0'[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ (root(t))\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') \to \\
\qquad\qquad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![e_1']\!]\ \kappa\ \rho\ \theta\ ), \ldots, (p_k, \mathcal{T}_0[\![e_k']\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Правила трансформации принимают на вход исходную программу и строят по ней с использованием определенных ранее правил помеченную систему переходов. После завершения трансформации помеченной системы переходов по ней строится новая, улучшенная за счет обобщений, программа.

Определение 1.2.13 Правила трансформации уровня n определены в формуле 9.

$$
\begin{array}{l}
\mathcal{T}_{n+1}[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{T}'_{n+1}[\![x \to (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (c, \mathbf{0}), (\#1, \mathcal{T}_{n+1}[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_{n+1}[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_c, \mathcal{T}_{n+1}[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (\lambda x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_\beta, \mathcal{T}_{n+1}[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \begin{cases}
(\kappa \bullet f) \to (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx^{\varnothing}_{\sigma} t \\
\mathcal{T}_{n+1}[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\
(\kappa \bullet f) \to (\tau_f, \mathcal{T}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise}
\end{cases} \\
\qquad\quad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\mathbf{let}, \mathcal{T}_{n+1}[\![e_1\{x \mapsto e_0\}]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
\tag{9}
$$

$$
\begin{array}{l}
\mathcal{T}'_{n+1}[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}'_{n+1}[\![(t\ e) \to (@, t), (\#1, \mathcal{T}_{n+1}[\![e]\!]\ \langle\rangle\ \rho\ \theta\ )]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}'_{n+1}[\![x \to (x, \mathbf{n+1})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\qquad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![e'_1]\!]\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Момент вызова правил обобщения из правил трансформации следует

рассмотреть подробнее. Из 9 следует, что трансформер уровня $n+1$ использует для обобщения помеченные системы переходов, накопленные в ρ, а также систему переходов, построенную с помощью трансформера уровня n. Правила обобщения для трансформера любого уровня могут быть вызваны только в случае, если текущее выражение является вызовом функции, т.е., было успешно сопоставлено с правилом трансформации для выражения $e = f$. Если $t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta$ является переименованием некоторой функции $t' \in \rho$, трансформация дальше не продолжается и осуществляется переход в конечное состояние 0, помеченный знаком τ_f. В случае, если t состоит с некоторой помеченной системой переходов $t' \in \rho$ в отношении вложения, запускается функция обобщения $G[t][t']$. Программа, построенная по обобщенной помеченной системе переходов передается в трансформер $n+1$ уровня для дальнейшей трансформации $\mathcal{T}_{n+1}[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\ \langle\rangle\ \rho\ \theta\ \varnothing$. Если ни один из предыдущих случаев не выполнился, помеченная система переходов t добавляется в ρ, затем вызываются правила трансформации для $\mathrm{T}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing$.

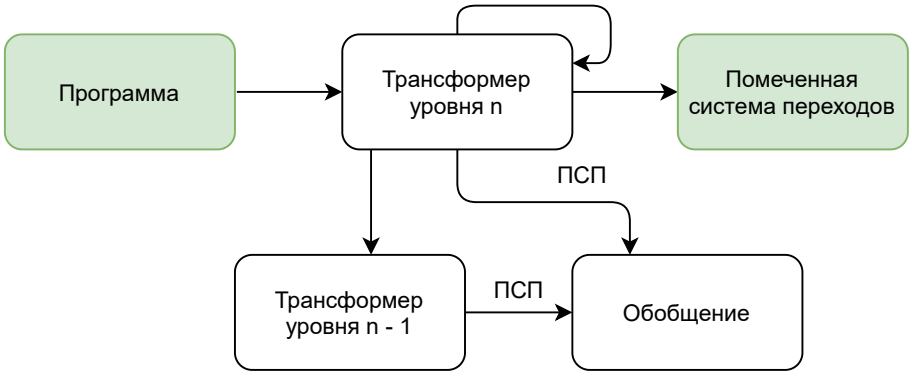

Рис. 7: Процесс работы правил трансформации

Для наглядности приведенные рассуждения представлены на рисунке 7: правила трансформации уровня n принимают на вход программу, используют для обобщения встречаемых помеченных систем переходов помеченные системы переходов, накопленные ранее, и трансформер уровня $n-1$, и в качестве результата выдают некоторую, потенциально содержащую обобщения, помеченную систему переходов.

С использованием правил трансформации можно построить иерар-

хию трансформированных помеченных систем переходов от уровня 0 до $n+1$. Переход от правил дистилляции уровня $n+1$ к уровню $n+2$ и вызов правил, определенных ранее — в частности, правил трансформации, осуществляют правила дистилляции, определенные в 1.2.14.

Определение 1.2.14 Алгоритм дистилляции выражен правилах, представленных в формуле 10.

$$
\begin{array}{l}
\mathcal{D}_n[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{D}'_n[\![x \to (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (c, \mathbf{0}), (\#1, \mathcal{D}_n[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{D}_n[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_c, \mathcal{D}_n[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{D}_n[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (\lambda x, \mathcal{D}_n[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_\beta, \mathcal{D}_n[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \left\{ \begin{array}{ll}
(\kappa \bullet f) \to (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx^{\varnothing}_{\sigma} t \\
\mathcal{D}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t^g]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ \{t^g\}\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\
where\ t^g = \mathcal{G}[\![t]\!][\![t']\!]\ \theta & \\
(\kappa \bullet f) \to (\tau_f, \mathcal{D}_n[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise}
\end{array} \right. \\
\qquad\quad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{D}_n[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\mathbf{let}, \mathcal{D}_n[\![e_1\{x \mapsto e_0\}]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{D}_n[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) \\
\\
\mathcal{D}'_n[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{D}'_n[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}'_n[\![(t\ e) \to (@, t), (\#1, \mathcal{D}_n[\![e]\!]\ \langle\rangle\ \rho\ \theta\ )]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}'_n[\![x \to (x, \mathbf{n+1})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{D}_n[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}'_n[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\qquad\qquad (\mathbf{case}, t), (p_1, \mathcal{D}_n[\![e'_1]\!]\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
\tag{10}
$$

Дистилляция программы является итеративным процессом — в случае, если в программе возможно осуществить обобщение, она обрабатывается дистиллятором следующего уровня до тех пор, пока в ней не останется обобщений, которые можно осуществить. Процесс работы алгоритма дистилляции представлен на рисунке 8.

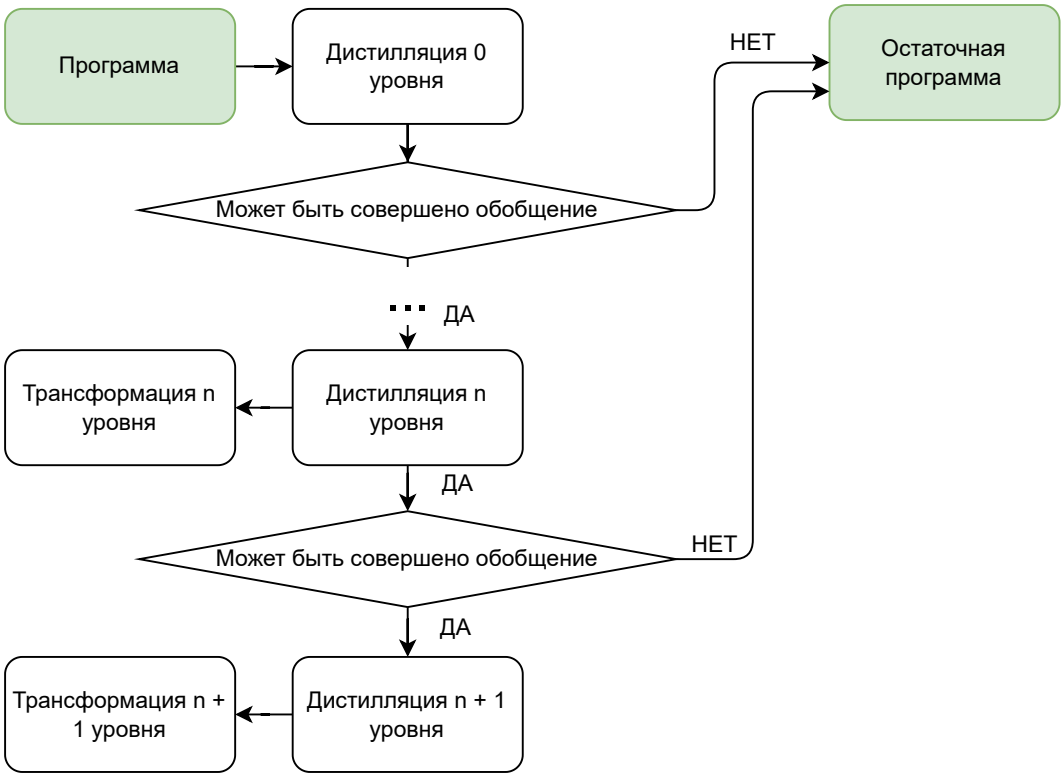

Рис. 8: Процесс работы правил дистилляции

1.3. Обзор существующего решения

В данном разделе будет представлена существующая архитектура проекта Distiller и некоторые детали ее реализации. Проект Distiller состоит из взаимодействующих друг с другом модулей *Main*, *Helpers*, *Trans*, *Term*, *Exception*, представленных на рисунке 9.

Существующая архитектура имеет недостаток, который заключается в том, что все описанные в 1.2 правила, выполняющие разные задачи алгоритма, в настоящей реализации хранятся в модулях *Trans* и *Term*. Это усложняет процесс изучения и написания кода, так как делает эти модули сочетающими сразу несколько этапов алгоритма дистилляции.

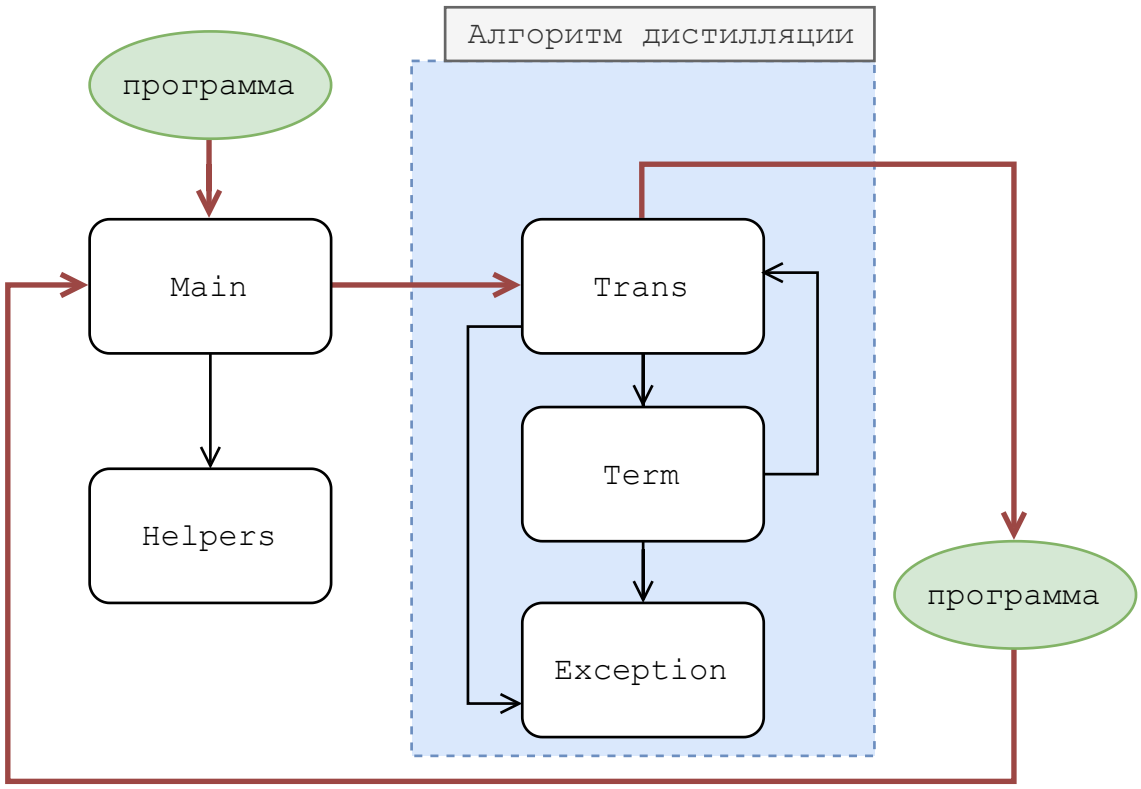

Рис. 9: Существующая архитектура проекта

Кроме того, это затрудняет процесс отладки и тестирования, так как настоящая реализация подхода дистилляции не точно соответствует алгоритму, изложенному в статье.

1.3.1. Особенности реализации

Проект получает на вход программу на простом функциональном языке, приведенном в формуле 1, совершает преобразования в соответствии с описанным алгоритмом дистилляции 1.2.14 и выдает результат — модернизированную программу, использующую меньшее количество промежуточных структур данных при вычислении. В качестве синтаксического анализатора для обработки текста программы был использован Parsec[5]. Результатом работы синтаксического анализа является исходная программа, представленная в виде терма, которая в этом формате используется в алгоритме дистилляции.

[5]Официальная документация и исходный код проекта могут быть найдены по ссылке https://hackage.haskell.org/package/parsec Дата последнего обращения 15.05.2022.

Рассмотрим более детально взаимодействие модулей друг с другом. Модуль *Main* реализует интерфейс общения с пользователем и предоставляет выполнение нескольких операций над поданной на вход программой:

- загрузить программу;
- показать исходный код программы;
- вычислить значение программы на введенных пользователем данных;
- представить программу в виде терма — результата синтаксического анализа;
- дистиллировать программу.

Для реализации приведенных операций и еще нескольких других был определен тип команды, приведенный в листинге 1.

Listing 1: Использующийся в проекте тип команды

```
type Directory = String
type OutputFileName = String
type InputFileName = String

data Command = Load InputFileName (Maybe Directory)
             | Prog
             | Term
             | Eval
             | Distill (Maybe OutputFileName)
             | Quit
             | Help
             | Unknown
```

Алгоритм дистилляции работает с программой, представленной в виде терма, тип которого представлен в листинге 2. Термы также используются в качестве меток в помеченной системе переходов, определенной в 1.2.7.

Listing 2: Использующийся в проекте тип терма

```
data Term = Free VariableName
    | Lambda VariableName Term
    | Con ConstructorName [Term]
    | Apply Term Term
    | Fun FunctionName
    | Case Term [(FunctionName,[VariableName],Term)]
    | Let VariableName Term Term
    | Unfold Term Term
    | Fold Term
    | Gen Term Term
```

Для контекста, определенного в 1.2.5, используется тип, приведенный в листинге 3. Имена функций, переменных и конструкторов, использующиеся в типе терма и типе контекста, имеют строковый тип и представлены в листинге 4.

Listing 3: Использующийся в проекте тип контекста

```
data Context = EmptyCtx
    | ApplyCtx Context Term
    | CaseCtx Context [(FunctionName,[VariableName],Term)]
  deriving Show
```

Listing 4: Типы аргументов

```
type VariableName = String
type ConstructorName = String
type FunctionName = String
```

В приведенных таким образом типах существует ряд серьезных недостатков. Так как тип терма используется как в качестве операнда в процессе синтаксического анализа, так и в качестве метки в помеченной системе переходов, это приводит к совмещению двух разных объектов в одном типе. Это усложняет написание и понимание кода в силу того, что некоторые конструкторы типа *Term* употребимы только в качестве

меток и не относятся к работе синтаксического анализа. Другим, более серьезным недостатком, является то, что в проекте нет типа, соответствующего помеченной системе переходов, что существенно осложняет реализацию сопоставления с образцом по переходам для всех правил алгоритма дистилляции, изложенных в статье [6] и приведенных в 1.2.

Недостатки существующих типов, описанные ранее, а также отсутствие отдельных модулей для каждого из этапов алгоритма, делает изучение, разработку и стабилизацию кода чрезвычайно трудоемкими. Модернизация существующих типов, добавление новых (например, типа для помеченной системы переходов), а также выделение различных частей алгоритма в отдельные модули, позволили бы существенно упростить отлаживание новой реализации, согласующейся с алгоритмом дистилляции, изложенным в 1.2 .

1.3.2. Обзор технологий, использующихся для тестирования существующего решения

В данном разделе представлен обзор методов тестирования планируемой реализации.

Так как на вход дистиллятору подается программа в файле, и результатом работы дистиллятора также является программа в файле, для предупреждения изменений в реализации, вносящих ошибки в существующую функциональность, возможно проверять набор файлов результирующих программ на совпадение с заранее заданными файлами. Также, так как программа, записанная файле, потенциально тоже может допускать ошибки, связанные с деталями предыдущей реализации, полезно проверять, что результирующая программа эквивалентна в своем поведении с исходной, то есть, проверять выполнение набора свойств, которые верны для исходной программы, на результате работы дистиллятора.

Описанные подходы позволяют проверить только корректность результатов работы алгоритма, но никак не отслеживают возможные ошибки в его реализации, которое осуществляет модульное тестирование. Поэтому при тестировании реализации алгоритма хотелось бы исполь-

зовать все три приведенных подхода:

- проверка того, что для одинаковых входных данных исходная программа и дистиллированная программа выдают одинаковый результат с помощью тестирования на основе свойств (*property-based testing*);
- проверка идентичности файлов модифицированной программы и ожидаемой программы (*golden testing*);
- проверка корректности реализации алгоритма с использованием модульного тестирования.

Инструмент, предоставляющий все описанные подходы, удобнее в использовании, чем подключение отдельной библиотеки или платформы для каждого из подходов, так как позволяет единообразный интерфейс доступа к различным видам тестирования, позволяя комбинировать несколько подходов в одном тесте. Для языка Haskell существует несколько таких инструментов: test-framework[6], Tasty[7], Hspec[8], HTF[9]. Так как инструмент Tasty осуществляет поддержку Hspec и описанных ранее подходов к тестированию, сравнение проводилось между Tasty, HTF и test-framework. Хотя test-framework является популярным инструментом для тестирования, его поддержка прекращена, и это является критичным недостатком этого инструмента по сравнению с Tasty и HTF.

По сравнению с HTF, Tasty предоставляет большее количество подходов к тестированию свойств. За счет интеграции библиотек smallcheck, quickcheck, leancheck Tasty предоставляет возможность исчерпывающего тестирования свойств, рандомизированного тестирования свойств,

[6]Официальная документация и исходный код проекта могут быть найдены по ссылке https://hackage.haskell.org/package/test-framework Дата последнего обращения 15.05.2022.

[7]Официальная документация и исходный код проекта могут быть найдены по ссылке https://hackage.haskell.org/package/tasty Дата последнего обращения 15.05.2022.

[8]Официальная документация и исходный код проекта могут быть найдены по ссылке https://hackage.haskell.org/package/hspec Дата последнего обращения 15.05.2022.

[9]Официальная документация и исходный код проекта могут быть найдены по ссылке https://hackage.haskell.org/package/HTF Дата последнего обращения 15.05.2022.

Тестовая платформа	Характеристики
test-framework	Очень распространен Не поддерживается
Tasty	*property-based testing*, *golden testing* Исчерпывающее тестирование свойств Рандомизированное тестирование свойств Тестирование перечислимых свойств
HTF	Рандомизированное тестирование свойств Не развитая поддержка
hspec	Интегрирован в Tasty

Таблица 1: Характеристики тестовых платформ для языка Haskell

тестирования перечислимых свойств, тогда как HTF интегрирует только библиотеку QUICKCHECK, предоставляя таким образом только возможность рандомизированного тестирования. HTF менее активно разрабатывается и обладает меньшим количеством обучающих статей и документации. Ввиду проведенных рассуждений в ходе обзора была выбрана платформа TASTY, предоставляющая все упомянутые виды тестирования.

Вышеприведенные рассуждения были собраны в таблицу 1, демонстрирующую характеристики рассмотренных тестовых платформ для языка Haskell.

2. Перепроектирование существующего решения

Одной из первостепенных задач исследования было перепроектирование существующей архитектуры дистиллятора. Обновленная архитектура проекта представлена на рисунке 10.

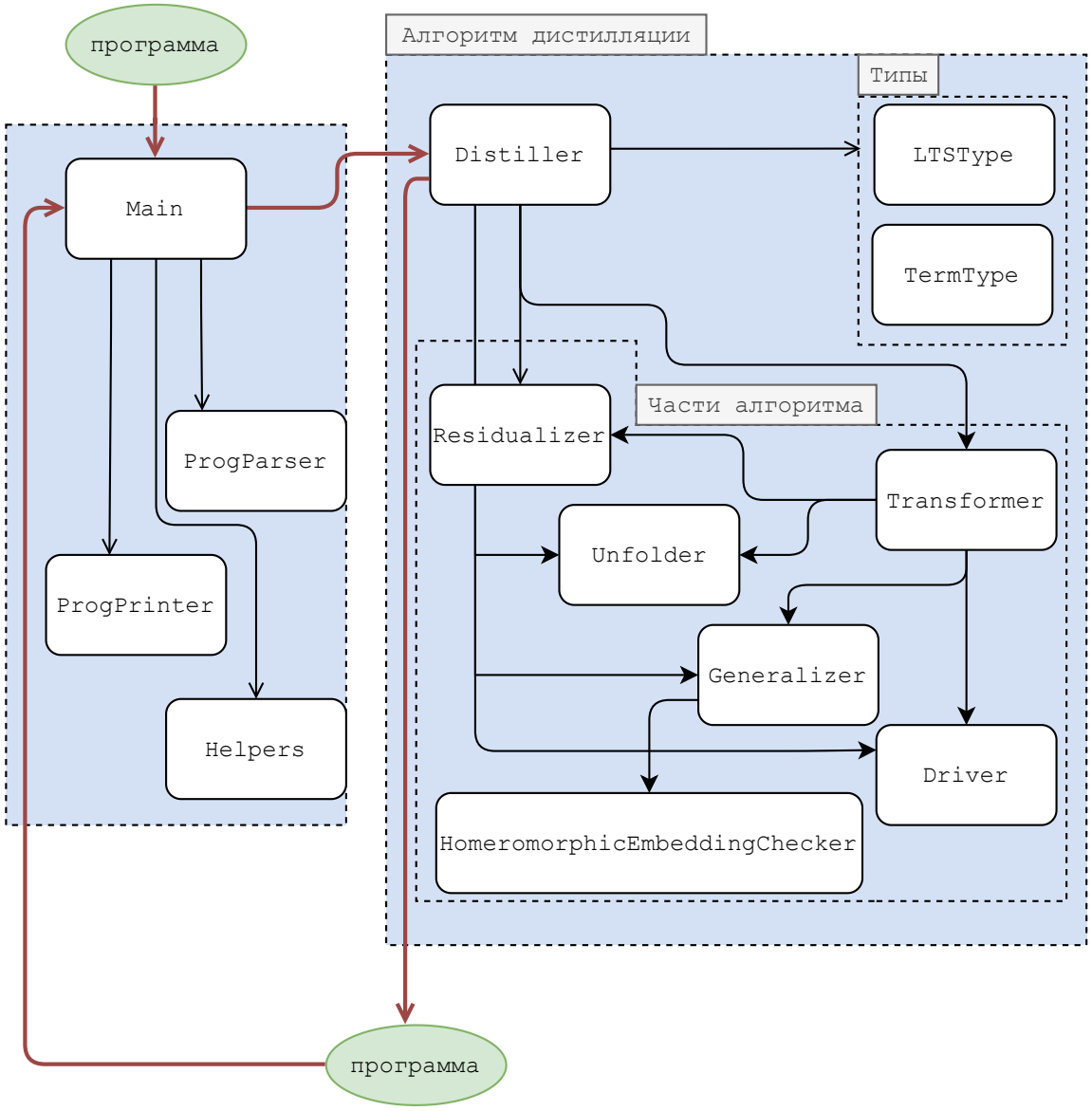

Рис. 10: Обновленная архитектура проекта

Диаграмма включает в себя два прямоугольника — слева интерфейс пользователя, который был отделен от модулей реализации, справа — модули, ответственные за реализацию алгоритма дистилляции. Синтаксический анализатор текстов входных программ, а также код, от-

ветственный за печать программы на экран и в файл, хранившиеся в модуле *Term* на диаграмме 9, были инкапсулированы в отдельные модули *ProgParser* и *ProgPrinter* соответственно.

Разбиение на модули различных шагов алгоритма дистилляции осуществлялось в соответствии с описанными в 1.2 правилами дистилляции: правилами построения помеченной системы переходов (модуль *Driver*), правилами обобщения (модуль *Generalizer*), правилами построения остаточной программы (модуль *Residualizer*), правилами трансформации (модуль *Transformer*), правилами, которые проверяют выполнения отношения вложения помеченных систем переходов (модуль *HomeomorphicEmbeddingChecker*).

С учетом недостатков существующих типов в проекте, упомянутых в 1.3.1, были добавлены новые типы: тип метки *Label* в помеченной системе переходов (см. листинг 5), тип помеченной системы переходов *LTS* (см. листинг 6), тип перехода *LTSTransition* (см. листинг 7). В множество типов, использующихся для обозначения аргументов, был добавлен тип имени аргумента конструктора, представленный в 8.

Listing 5: Введенный тип метки

```
data Label = Con' ConstructorName
     | ConArg' ConstructorArgumentName
     | X' VariableName
     | Lambda' VariableName
     | Unfold' FunctionName
     | UnfoldBeta'
     | UnfoldCons' ConstructorName
     | Case'
     | CaseBranch' ConstructorName [VariableName]
     | Let'
     | LetX' VariableName
     | Apply0'
     | Apply1' deriving (Eq, Show)
```

Listing 6: Введенный тип помеченной системы переходов

```
data LTS = Leaf   | LTS LTSTransitions
        deriving (Eq, Show)
```

Listing 7: Введенный тип перехода в помеченной системе переходов

```
data LTSTransitions = LTSTransitions Term [(Label, LTS)]
```

Listing 8: Типы аргументов

```
type VariableName = String
type ConstructorName = String
type FunctionName = String
type ConstructorArgumentName = String
```

Также, с учетом введенного типа *Label*, был изменен тип *Term* (см. листинг 9) — в нем были оставлены только конструкторы, использующиеся при синтаксическом анализе программы.

Listing 9: Обновленный тип терма

```
data Term = Free Variable
    | Lambda VariableName Term
    | Con ConstructorName [Term]
    | Apply Term Term
    | Fun FunctionName
    | Case Term [(FunctionName,[Variable],Term)]
    | Let VariableName Term Term
    deriving Show
```

Добавленные типы упростили реализацию сопоставлений с образцом, используемых в алгоритме, а также облегчили изучение и модификацию алгоритма, так как позволили отказаться от использования типа *Term* для нескольких сущностей сразу.

3. Разработка дистиллятора

В статьях, в которых изложен алгоритм дистилляции, подходы к реализации никак не рассматриваются. При реализации этого алгоритма в рамках выпускной квалификационной работы был найден ряд неточностей в описании алгоритма, приводящих к проблемам при его реализации. Поэтому с целью облегчения тестирования и стабилизации алгоритма дистилляции при интеграции его в проект дистиллятора было решено упростить некоторые шаги алгоритма. В этой главе будут рассмотрены произведенные упрощения реализации, а также приведены основные найденные во время реализации неточности в описании алгоритма и решения, примененные в реализации, для устранения этих неточностей.

3.1. Упрощение шагов алгоритма дистилляции

В качестве модуля, стабилизация которого может быть предварительно опущена с целью обеспечения стабильности остальных модулей алгоритма при интеграции его в проект дистиллятора, был взят модуль *Generalizer*. На диаграмме 10 видно, что данный модуль может быть вызван только модулями трансформации или дистилляции, поэтому комплексное тестирование алгоритма дистилляции может проводиться в два этапа. На первом этапе можно тестировать только множество программ, на которых функции этого модуля не вызываются, что позволит протестировать совместную работу всех остальных модулей и процесса работы дистиллятора в целом. На втором этапе можно добавить к тестированию множество программ, вызывающих различные модуля *Generalizer*. Второй этап комплексного тестирования было решено проводить за пределами выпускной квалификационной работы.

3.2. Реализация нового алгоритма дистилляции

Алгоритм дистилляции использует множество параметров при вычислении, и проблема необходимости сохранения и распространения

тех или иных параметров на различных этапах работы программы нуждается в исследовании. При этом области распространения параметров могут быть разными: параметры могут распространяться между различными уровнями правил трансформации или дистилляции, могут распространяться между ветками помеченной системы переходов в процессе отрабатывания правил алгоритма, могут распространяться между правилами различных модулей, например, между правилами построения остаточной программы и трансформации. Для некоторых параметров, не освещаемых подробно в статье, будут рассмотрены и пояснены области распространения, а также приведены подходы к реализации их распространения.

3.2.1. Проблема распространения информации при реализации алгоритма дистилляции: удаление дублирующихся обобщений

В процессе работы правил обобщения, исходя из информации, присутствующей в обрабатываемой ветке, создаются помеченные системы переходов — обобщения. Так как в правилах обобщения 1.2.11 не предусмотрен обмен информацией между ветками помеченной системы переходов, при реализации алгоритма была выявлена проблема появления дублирующихся обобщений — пар обобщений вида $\{x \to t\}$ и $\{x \to t'\}$, либо $\{x \to t\}$ и $\{x' \to t\}$, то есть, имеющих одинаковую правую или левую часть. В случае совпадающих левых частей в процессе дистилляции программы возникает коллизия имен переменных, в случае совпадающих правых частей накапливаются разные с точки зрения алгоритма дистилляции обобщения $\{x \to t\}$ и $\{x' \to t\}$, ответственные за одну и ту же помеченную систему переходов t. Для решения данной проблемы был разработан и реализован алгоритм действий, для изложения которого требуется ввести некоторые обозначения.

Пусть каждое обобщение, принадлежащее результирующей помеченной системе переходов t^g, имеет вид $\{x \to t\}$, где x — переменная в обобщенной помеченной системе переходов, а t — извлеченная из исходной помеченной системы переходов подсистема). Тогда для избежания

возникновения дублирующихся обобщений при их объединении следует осуществлять следующую последовательность действий:

- накопленные обобщения из каждой ветки добавляются в структуру данных "Словарь", в которой ключом является правая часть обобщения — помеченная система переходов t, а значением — пара (x, t^g);
- все переменные x обобщения $\{x \to t\}$, входящие в состав пар (x, t^g), ассоциированных с t, перенумеровываются для избежания коллизий имен переменных;
- каждая результирующая помеченная система переходов t^g, имеющая набор обобщений вида $\{x \to t\}$, обращается в хранящую обобщения структуру данных "Словарь" по ключу t и получает новое название переменной x;
- каждая результирующая помеченная система переходов t^g, получившая x^i — новое название переменной x, осуществляет переименование используемой переменной x в x^i.

Приведенная последовательность действий применяется для всех правил обобщения, в которых обобщение ведется в нескольких ветках одновременно, т.е., для обобщения конструкторов, **case**, **let**, операции применения функции. Хотя интеграционное тестирование реализации модуля *Generalizer* не проводилось ввиду того, что его реализация была упрощена, для проверки данного алгоритма было проведено модульное тестирование.

3.2.2. Проблема распространения информации при реализации алгоритма дистилляции: накопление новых функций

Во время выполнения правил построения остаточной программы образовываются новые функции. Рассмотрим подробнее следующее правило, которое выполняется случае, если во время правил построения

остаточной программы в помеченной системе переходов была встречена функция:

$$\mathcal{R}'[\![e \to (\tau_f, t)]\!]\ \varepsilon = \begin{cases} e'\theta, if \exists (e' = e'') \in \varepsilon \bullet e \equiv e''\theta \\ f\ x_1 \ldots x_n \textbf{where} f' = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \quad \text{otherwise}\ \ (f'\ \text{ is }\ \text{fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases}$$

Выполнение данного правила позволяет обозначить вызов встреченной функции f за новую функцию f' и в дальнейшем заменять выражения, являеющиеся вызовом функции f с точностью до переименования аргументов, на вызов новой функции f'. Это оптимизирует процесс построения программы, так как позволяет не обрабатывать одинаковые с точностью до переименования подсистемы в помеченной системе переходов, построенные для вызова функции f. В качестве примера образования новой функции f можно рассмотреть результат дистилляции программы *append xs* []. Функция *append xs ys* определена следующим образом:

```
append xs ys = case xs of
                 Nil  →  ys
               | Cons(x,xs')  →  Cons(x,append xs' ys)
```

В процессе дистилляции рассматриваемой программы вместо аргументов xs и ys будут подставлены их актуальные значения, поэтому для *append xs* [] тело функции будет иметь вид:

```
append xs [] = case xs of
                 Nil  →  []
               | Cons(x,xs')  →  Cons(x,append xs' [])
```

Приведенное выше правило для функционального вызова в процессе построения программы по помеченной системе переходов создаст новую функцию f', имеющую следующее определение:

```
f' xs = case xs of
          Nil  →  []
        | Cons(x,xs')  →  Cons(x,f' xs')
```

Согласно правилу, функция f' имеет в качестве определения функции программу, соответствующую помеченной системе переходов, построенной для $append\ xs\ []$. Аргументами этой функции являются все свободные переменные помеченной системы переходов, построенной для $append\ xs\ []$. Так как в помеченной системе переходов, построенной для $append\ xs\ []$ свободной переменной является только xs (т.к. в качестве ys было ранее подставлено значение $[]$, и данная свободная переменная больше не используется в помеченной системе переходов), функция f' использует только один аргумент — xs.

Хотя согласно определению 6, в правила построения программы по помеченной системе переходов множество определений функций не передается, введенная функция $f\ x$ в таком случае должна быть добавлена в множество определений функций Δ, использующееся в правилах трансформации 8, 9 и дистилляции 10, так как построенная программа будет содержать в себе вызовы этой функции. Таким образом, аккумулятор множества определений функций должен быть распространен между правилами трансформации, дистилляции и правилами построения программы по помеченной системе переходов. Его распространение было реализовано через использование в качестве параметра в правилах построения программы по помеченной системе переходов аккумулятора определений функций Δ, используемом в 8, 9 и 10. Этот аккумулятор объединил в себе хранение функциональных вызовов — выражений вида $f\ xs = append\ xs\ []$, которые предполагалось хранить в и хранение определений функций.

3.2.3. Проблема неконсистентного введения новых функций при реализации алгоритма дистилляции

Проблема введения новых функций, описанная ранее, содержит в себе не только проблему распространения аккумулирующего множества функций между правилами трансформации, дистилляции и правилами построения программы по помеченной системе переходов. После того, как остаточная программа построена, информация о новых функциях не передается в аккумулятор уже построенных помеченных

систем переходов, и в этих системах переходов используется функциональный вызов, который был ассоциирован с новым функциональным вызовом во время построения остаточной программы. Неконсистентность в использовании разных функциональных вызовов возникает во время выполнения правил трансформации или дистилляции любого уровня для функции. Не умаляя общности, для иллюстрации этой проблемы рассмотрим правило трансформации n уровня, которое представлено в 11.

$$
\begin{array}{l}
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_\beta, \mathcal{T}_{n+1}[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \left\{ \begin{array}{ll}
(\kappa \bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx^{\varnothing}_{\sigma} t \\
\mathcal{T}_{n+1}[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!] \diamondsuit\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\
(\kappa \bullet f) \rightarrow (\tau_f, \mathcal{T}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!] \diamondsuit\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise}
\end{array} \right. \\
\qquad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta
\end{array}
$$

Рис. 11: Правило трансформации, иллюстрирующее неконсистентное введение новых функций

В качестве примера может быть рассмотрена упомянутая ранее функция $append\ xs\ ys$. В правилах трансформации уровня $n+1$ функция $append$ с контекстом $k = (\bullet\ xs)\ ys$ сопоставится с правилом трансформации для функции, приведенным выше. В результате сопоставления с образцом по помеченной системе переходов $t = T_n[append]\ ((\bullet\ xs)\ ys)\ \varnothing\ \theta\ \Delta$ будет построена программа t' с использованием правил построения программы по помеченной системе переходов. В этой программе t' будет использоваться функциональный вызов $f'\ xs$, обозначающий вызов функции $append\ xs\ []$, введение которого подробно обсуждалось в предыдущей секции. Таким образом, в процессе выполнения рассматриваемого правила трансформации образуется следующий переход в помеченной системе переходов:

$$(append\ xs\ []) \rightarrow (\tau_{append}, T_{n+1}[\![\mathcal{U}[\![t']\!]]\!] \diamondsuit\ (\rho \cup \{t\})\ \theta\ )$$

где в правой части перехода для обозначения функционального вызова $append\ xs\ []$ используется $f'\ xs$, а в левой части (и в предшествующих переходах) продолжает использоваться $append\ xs\ []$. Информация о том, что $append\ xs\ []$ должен быть заменен на $f'\ xs$ также не распространяется и среди протрансформированных систем переходов, которые хранятся в аккумуляторе ρ. Корректное обновление уже построенных помеченных систем переходов новыми функциональными вызовами важно, так как в противном случае может препятствовать выполнению отношения гомеоморфного вложения между двумя помеченными системами переходов. Например, хотя $f'\ xs$ обозначает $append\ xs\ []$, эти конструкции не являются гомеоморфно вложенными. Остаточная программа, построенная по помеченной системе переходов, соответствующей протрансформированному выражению $append\ xs\ []$, которая использует $f'\ xs$ для обозначения $append\ xs\ []$, не отличается от программы, соответствующей $append\ xs\ []$ с точностью до обозначений $f'\ xs = append\ xs\ []$. Но помеченная система переходов, использующая $append\ xs\ []$ и помеченная система переходов, использующая $f'\ xs$, не будут состоять в отношении гомеоморфного вложения, хотя известно, что $f'\ xs = append\ xs\ []$. Таким образом, при дальнейшей трансформации помеченной системы переходов, использующей $append\ xs\ []$, новая протрансформированная помеченная система переходов, в которой фигурирует $f'\ xs$, не будет применена, т.к. не выполнится отношение гомеоморфного вложения.

4. Тестирование дистиллятора

В процессе реализации упрощенного алгоритма дистилляции в проект было добавлено более 120 различных тестов, а также подключена непрерывная интеграция с использованием GitHub Actions, которая собирает версию проекта, хранящуюся в ветке с разрабатываемым алгоритмом, и запускает имеющиеся в нем тесты. В данной главе будут рассмотрены подходы, которые были применены к тестированию упрощенного алгоритма дистилляции.

4.1. Интеграционное тестирование

Модули *HomeomorphicEmbeddingChecker* и *Driver* не имеют зависимостей от других модулей реализации алгоритма и поэтому были протестированы с помощью модульных тестов, однако остальные модули реализации алгоритма на диаграмме 10 зависят друг от друга, и поэтому нуждаются в интеграционном тестировании.

Так, модули *Residualizer* и *Generalizer* используют в своей работе помеченные системы переходов, построенные в процессе работы правил трансформации. Так как помеченные системы переходов трудно задавать вручную, для тестирования этих модулей был использован модуль *Driver*, строящий тестируемые помеченные системы переходов. Таким образом, для модулей *Residualizer* и *Generalizer* было проведено интеграционное тестирование с использованием модуля *Driver*.

Тестирование модулей *Distiller* и *Transformer* представило наибольшую сложность, так как из диаграммы архитектуры на рисунке 10 видно, что в своей работе они используют почти все остальные модули алгоритма дистилляции. Так как на момент прохождения выпускной квалификационной работы в проекте уже существовал набор программ, которые могли бы быть использованы как входные программы для дистиллятора, для создания интеграционного тестирования модулей *Distiller* и *Transformer* было решено отобрать ряд программ из этого набора.

В проекте имеется три группы входных программ, которые отлича-

ются друг от друга типом входных аргументов. Эти программы являются вызовами функций, сосредоточенных в файлах *Bool*, *Nat*, *NatList* и представлены на рисунке 12. Файлы отмечены синим цветом, аргументы, используемые функциями, определенными в файле — зеленым. Функции, определенные в файле *NatList*, используют функции, определенные в файлах *Bool* и *Nat*.

Функции, сосредоточенные в файле *Bool*, используют в качестве аргументов булевы значения $True$ и $False$, отмеченные на рисунке 12 зеленым цветом. Программы, определенные в этом файле есть простейшие булевы операции над аргументами x и y, которые могут принимать значения $True$ или $False$. Например, в файле *Bool* имеются определения функций, вычисляющих логическую конъюнкцию и дизъюнкцию переменных x и y (функции *and* и *or* соответственно), определение функции, вычисляющей значение импликации для x и y (функция *implies*) и другие.

Функции, сосредоточенные в файле *Nat*, используют в качестве аргументов числа арифметики Пеано — $Zero$ и $Succ(x)$, отмеченные на рисунке 12 зеленым цветом. Для этих чисел в файле определены простейшие операции — функция сложения двух чисел (функция *plus*), функция вычитания двух чисел (функция *minus*) и другие.

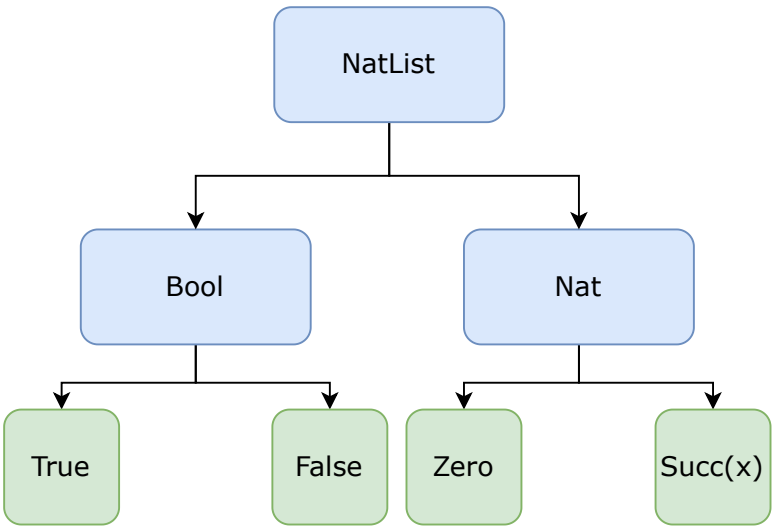

Рис. 12: Процесс проверки сохранения формата данных

Функции, сосредоточенные в файле *NatList*, оперируют над списками значений булевого типа или списками чисел арифметики Пеано. Они используют функции, определенные в файлах *Bool* и *Nat*. Например, в файле *NatList*, определена функция *append*, конкатенирующая списки xs ys, функция $nrev$, переворачивающая список xs, функция map, применяющая указанную функцию f к каждому элементу списка xs, а также ряд других функций.

Существует ряд достаточно простых программ, заключающихся в вызове функций, определенных в файлах *Bool*, *Nat*, *NatList*, с определенными параметрами. Например, функция *plusplus* вызывает функцию сложения чисел Пеано *plus* для аргумента x, подставленного дважды, то есть, реализует сложение аргумента x с самим собой. Также есть функция $append\_Nil$, вызывающая функцию конкатенации списков xs и ys для ys, имеющего значение пустого списка.

Программы f, fg и $getSndListRec$ были написаны специально для интеграционного тестирования и не используют описанные ранее функции. Программа f является вызовом бесконечнорекурсивно определенной функции $f\ x\ =\ f\ x$, а программа $fg\ x$ является вызовом бесконечнорекурсивно определенной функции $f\ x\ = g\ x$, где $g\ x\ = f\ x$. Эти программы позволили проверить работу алгоритма дистилляции на бесконечнорекурсивно определенных функциях. Засчет своей простоты они легко могут быть отлажены и поэтому использовались в качестве первых тестов для алгоритма дистилляции. Программа $getSndListRec$ для двух списков xs и ys рекурсивно разбирает список xs, и, завершив разбор, возвращает список ys. Эта программа является простейшей программой по работе со списком — она содержит только оператор *case* для разбора списка и рекурсивный вызов без каких-либо дополнительных конструкций, фигурирующих в ранее приведенных программах, работающих со списками (как, например, конструктор в функции для конкатенации списков *append*, который используется для создания нового списка). Ввиду своей простоты эта программа являлась вспомогательным тестом к проведению тестирования более сложных программ, работающих со списками, как, например, *append*, *map* и другие.

Bool	Other
not	plus
and	plusplus
iff	append
or	getSndListRec
implies	f
eqBool	fg

Таблица 2: Набор программ для интеграционного тестирования

В итоге тестирование модулей *Distiller* и *Transformer* производилось с использованием программ из семейства *Bool*, а также нескольких других программ. Подобранные программы дистиллировались, и результирующая помеченная система переходов сравнивалась на равенство с эталонной системой переходов, построенной заранее. Все использованные для интеграционного тестирования входные программы представлены в таблице 2.

Таким образом, была протестирована работа алгоритма дистилляции на простейших программах, таких как программа вычисления $and\ x\ y$ для булевых значений x и y, программа $plus\ \ x\ \ y$, осуществляющая операцию вычитания в терминах арифметики Пеано (т.е., x, y могут принимать значения $S...S(0)$ или 0), программа $append\ xs\ ys$, осуществляющая конкатенацию двух списков, и т.д.

4.2. Функциональное тестирование

Проект дистиллятор имеет интерфейс общения с пользователем, принимающий на вход файл программы, и передающий на выход файл продистиллированной программы. Помимо интеграционного тестирования реализации упрощенного алгоритма дистилляции важно убедиться, что с разработкой новой реализации сохранился формат входных и выходных программ, используемый интерфейсом дистиллятора для передачи алгоритму дистилляции. Проверка сохранности формата данных была организована посредством запуска алгоритма дистилляции на загруженных из файлов программах и последующей проверке на

идентичность файлов продистиллированной алгоритмом программы и ожидаемой программы *golden testing*. Диаграмма, иллюстрирующая процесс проверки, представлена на рисунке 13.

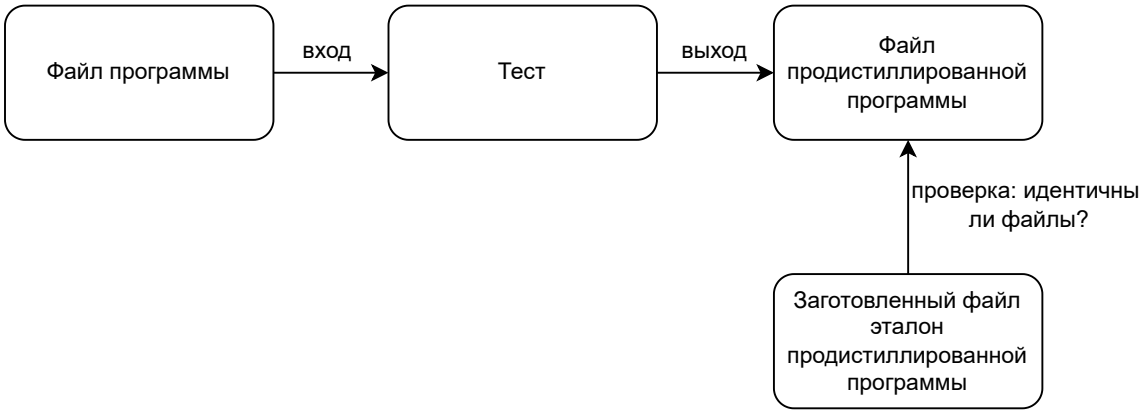

Рис. 13: Процесс проверки сохранения формата данных

Такое тестирование позволило проверить, что формат входных и выходных программ не поменялся. В качестве тестовых программ были взяты программы, приведенные в разделе интеграционного тестирования, а также несколько других, представленных на таблице 3.

Bool	Nat	Other
not	minus	append
and	plus	append_Nil
iff	mul	qrev
or	eqNat	f
implies	plusplus	fg
eqBool	gt	map

Таблица 3: Набор программ для функционального тестирования

4.3. Тестирование на основе свойств программ

Если для простейших программ, используемых в интеграционном и функциональном тестировании, довольно легко вручную оценить, что

поведение продистиллированной программы не отличается от поведения исходной программы, то для более сложных программ проверить это вручную может быть крайне трудно. Для автоматизации проверки того, что поведение продистиллированной программы ничем не отличается от поведения исходной, использовался подход тестирования свойств программ.

Свойством программы считается ее значение Y для входных данных X. В случае, если для оригинальной программы свойство выоплняется, а для продистиллированной — нет, значит, алгоритм дистилляции отработал неправильно, так как две программы перестали иметь одинаковое поведение.

В качестве примера можно рассмотреть две программы — исходную и продистиллированную программы для булевой функции *iff*, представленные в выкладках 10, 11.

Listing 10: Программа *iff x y*

```
main = iff x y;
iff x y
    = case x of
        True -> y
        | False -> not y;
not x
    = case x of
        True -> False
        | False -> True
```

Для любых значений x и y эти программы будут выдавать одинаковые результаты, т.к. вторая программа отличается от первой только тем, что в нее подставлено определение функции *not* для значения y. С другой стороны, если в алгоритме дистилляции произошла ошибка, и в тесте сравниваются программы, представленные на рисунках 10, 12 (ошибка алгоритма заключается в неправильной подстановке определения функции *not*), то найдутся значения x и y, на которых эти программы выдадут разный результат. Для рассматриваемых программ

достаточно взять $x = False$ и $y = True$, тогда $iff\ False\ True = False$, и $f''\ False\ True = True$. В общем случае тестирование на основе свойств производится проверкой поведений программ для диапазонов значений.

Listing 11: Корректный результат дистилляции *iff x y*

```
main = f'' x y;

f'' x y
    = case x of
        True -> y
        | False -> (case y of
                      True -> False
                      | False -> True);
```

Данный вид тестирования не слишком широко использовался, так как для проверки упрощенной версии алгоритма были подобраны достаточно простые программы, эквивалентность поведений которых могла быть легко установлена вручную.

Listing 12: Результат дистилляции *iff x y*, содержащий ошибку

```
main = f'' x y;

f'' x y
    = case x of
        True -> y
        | False -> (case y of
                      True -> True
                      | False -> True);
```

В дальнейшем, при доработке алгоритма и расширении его тестового покрытия более сложными программами, этот подход планируется использовать как единственно возможное средство проверки, не требующее подробного изучения результата дистилляции, и использующее только исходную программу и различные входные данные, понятные любому пользователю и тестировщику.

Заключение

В ходе выпускной квалификационной работы в проекте Distiller[10] были достигнуты следующие результаты.

- Интерфейс общения с пользователем был отделен от существующей реализации и разработана архитектура для новой реализации алгоритма. Различные шаги алгоритма выделены в отдельные модули, добавлены новые типы — тип метки и тип помеченной системы переходов для реализации сопоставления с образцом в алгоритме.

- В соответствии с разработанной архитектурой реализован и интегрирован в проект упрощенный алгоритм дистилляции на языке Haskell. Реализация модуля Generalizer была упрощена с целью обеспечения стабилизации остальных модулей алгоритма.

- В рамках подготовки к тестированию алгоритма была настроена тестовая инфраструктура – добавлена непрерывная интеграция исходного кода проекта с использованием Github Actions, а так же настроена и подключена тестовая платформа Tasty.

- Проведено тестирование полученной реализации и выявлены неточности в описании алгоритма, приводящие к проблемам, актуальным для любой его реализации:

  – Неконсистентное введение новых функций алгоритмом;

  – Некорректное распространение информации между разными уровнями алгоритма.

  Все изложенные проблемы для полученной реализации были решены.

[10]Официальная документация и исходный код проекта могут быть найдены по ссылке https://github.com/YaccConstructor/Distiller/tree/draft-arch, пользователь katevi. Дата последнего обращения 15.05.2022.

Таким образом, в рамках выпускной квалификационной работы была реализована упрощенная версия алгоритма дистилляции и проведено комплексное тестирование полученного дистиллятора.

Список литературы

[1] Chris Leary Todd Wang. XLA: TensorFlow, compiled. — https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. — 2017.

[2] Coutts Duncan, Leshchinskiy Roman, and Stewart Don. Stream Fusion: From Lists to Streams to Nothing at All // SIGPLAN Not. — 2007. — oct. — Vol. 42, no. 9. — P. 315–326. — Access mode: https://doi.org/10.1145/1291220.1291199.

[3] Davis Timothy A. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra // ACM Trans. Math. Softw. — 2019. — dec. — Vol. 45, no. 4. — Access mode: https://doi.org/10.1145/3322125.

[4] Buluç Aydin, Mattson Tim, McMillan Scott, Moreira José, and Yang Carl. Design of the GraphBLAS API for C // 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2017. — P. 643–652.

[5] Hamilton Geoff. Extracting the Essence of Distillation. — 2009. — 06. — P. 151–164.

[6] Hamilton Geoff. A Hierarchy of Program Transformers. — 2012. — 01. — Access mode: https://www.researchgate.net/publication/229062264_A_Hierarchy_of_Program_Transformers.

[7] Hamilton Geoffrey. The Next 700 Program Transformers. — 2021. — 2108.11347.

[8] Jones Neil D., Gomard Carsten K., and Sestoft Peter. Partial Evaluation and Automatic Program Generation. — USA : Prentice-Hall, Inc., 1993. — ISBN: 0130202495.

[9] Klimov Andrei. An approach to Supercompilation for Object-Oriented Languages: the Java Supercompiler Case Study. — 2008. — 01.

[10] Klimov Andrei. A Java Supercompiler and Its Application to Verification of Cache-Coherence Protocols. — 2010. — 01. — P. 185–192.

[11] Lattner Chris, Pienaar Jacques, Bondhugula Uday, Riddle River, Cohen Albert, Shpeisman Tatiana, Davis Andy, Vasilache Nicolas, and Zinenko Oleksandr. MLIR: A Compiler Infrastructure for the End of Moore's Law. — 2020. — 02.

[12] Ribeiro Francisco, Saraiva João, and Pardo Alberto. Java Stream Fusion: Adapting FP Mechanisms for an OO Setting // Proceedings of the XXIII Brazilian Symposium on Programming Languages. — New York, NY, USA : Association for Computing Machinery. — 2019. — SBLP 2019. — P. 30–37. — Access mode: https://doi.org/10.1145/3355378.3355386.

[13] Kiselyov Oleg, Biboudis Aggelos, Palladinos Nick, and Smaragdakis Yannis. Stream Fusion, to Completeness // CoRR. — 2016. — Vol. abs/1612.06668. — arXiv : 1612.06668.

[14] Sørensen Morten, Gluck Robert, and Jones Neil. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. — 1997. — 01.

[15] Sørensen Morten Heine. Turchin's Supercompiler Revisited - An operational theory of positive information propagation. — 1996. — Access mode: https://pat.keldysh.ru/~roman/doc/Turchin/1996-Soerensen--Turchin's_Supercompiler_Revisited--An_operational_theory_of_positive_information_propagation.pdf.

[16] Abadi Martin, Barham Paul, Chen Jianmin, Chen Zhifeng, Davis Andy, Dean Jeffrey, Devin Matthieu, Ghemawat Sanjay, Irving Geoffrey, Isard Michael, Kudlur Manjunath, Levenberg Josh, Monga Rajat, Moore Sherry, Murray Derek G., Steiner Benoit, Tucker Paul, Vasudevan Vijay, Warden Pete, Wicke Martin, Yu Yuan,

and Zheng Xiaoqiang. TensorFlow: A system for large-scale machine learning // 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). — 2016. — P. 265–283. — Access mode: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[17] Turchin Valentin F. The Concept of a Supercompiler // ACM Trans. Program. Lang. Syst. — 1986. — jun. — Vol. 8, no. 3. — P. 292–325. — Access mode: https://doi.org/10.1145/5956.5957.

[18] Wadler Philip. Deforestation: transforming programs to eliminate trees // Theoretical Computer Science. — 1990. — Vol. 73, no. 2. — P. 231–248. — Access mode: https://www.sciencedirect.com/science/article/pii/030439759090147A.

[19] Yang Carl, Buluç Aydin, and Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU // CoRR. — 2019. — Vol. abs/1908.01407. — arXiv : 1908.01407.