Труды лаборатории языковых инструментов

Выпуск 5

Санкт-Петербург, 2017

Предисловие

Здесь будет аннотация

куратор лаборатории Д.Булычев

Оглавление

Применение парсер-комбинаторов для разбора булевых грамматик

Курбатова Зарина Идиевна

Санкт-Петербургский государственный университет
zarina.kurbatova@gmail.com

Аннотация Булевы грамматики являются расширением контекстно-свободных грамматик, которое позволяет использовать конъюнкцию и отрицание в правилах вывода. В рамках данной работы была разработана принтер-комбинаторная библиотека для разбора булевых грамматик. Для получения приемлимой производительности и поддержки левой рекурсии в библиотеке были использованы техники мемоизации и CPS (continuation-passing style).

Введение

Языки программирования имеют строгую синтаксическую структуру, которую можно успешно описать с использованием формальных грамматик. Согласно иерархии Н.Хомского [8], формальные грамматики делятся на 4 вида: неограниченные, контекстно-свободные, контекстно-зависимые и регулярные. Как известно, с помощью контекстно-свободных грамматик может быть описан весьма узкий класс языков. Существует расширение контекстно-свободных грамматик – булевы грамматики, предложенные А.Охотиным [1]. Булевы грамматики более выразительны, поскольку позволяют использовать конъюнкцию и отрицание в правых частях правил вывода.

Задачу синтаксического анализа можно сформулировать как задачу определения принадлежности слова некоторому

языку. Иными словами, синтаксический анализатор принимает на вход строку и определяет, может ли грамматика задаваемого языка порождать введенную строку. Одним из популярных подходов к реализации синтаксических анализаторов является парсер-комбинаторная техника: анализаторы представляются функциями высших порядков [5]. Парсер-комбинаторы – это функции высшего порядка, принимающие в качестве параметров анализаторы и возвращающие в качестве результата анализатор. Анализатор, в свою очередь, принимает в качестве входа строку и возвращает некоторый результат. При данном подходе определяются базовые комбинаторы, а затем с их помощью определяются более сложные. Например, в качестве базовых комбинаторов можно использовать комбинатор последовательного применения и комбинатор альтернативного применения. Основной проблемой наивной реализации подхода является невозможность использования левой рекурсии. Существуют различные реализации данной техники, поддерживающие левую рекурсию. Например, в работе [4] решение основано на алгоритме обобщенного синтаксического анализа GLL.

Синтаксический анализ находит применение во многих областях. В биоинформатике одной из основных задач является поиск в геноме особых участков, кодирующих белок, тРНК и др. Геном представляет собой последовательность нуклеотидов или, другими словами, строку над алфавитом $\{A, C, G, T\}$. Искомые подстроки генома могут быть описаны с помощью контекстно-свободной грамматики. Таким образом, необходимо находить все строки, обладающие свойством выводимости. Например, вторичная структура РНК может быть описана с помощью контекстно-свободной грамматики, поскольку является своего рода языком палиндромов. Поэтому методы синтаксического анализа применимы при решении задач биоинформатики [6].

1 Постановка задачи

Целью данной работы является изучение применимости парсер-комбинаторной техники для разбора булевых грамматик. Для ее достижения были поставлены следующие задачи:

- реализация эффективной парсер-комбинаторной библиотеки для разбора булевых грамматик с поддержкой левой рекурсии;
- апробация библиотеки на примере контекстно-свободной грамматики, булевой грамматики и грамматики из биоинформатики;
- сравнение с существующими решениями.

2 Обзор предметной области

В данной главе определены основные понятия и приведено описание техник, использованных при реализации парсер-комбинаторной библиотеки.

2.1 Парсер-комбинаторы

В функциональном программировании представление анализаторов функциями и определение функций высших порядков (комбинаторов) является популярным подходом к построению нисходящих синтаксических анализаторов. При данном подходе определяются базовые комбинаторы и примитивные анализаторы, например, для разбора пустой строки и терминального символа.

Рассмотрим несколько таких комбинаторов.

Листинг 1.1: Комбинатор seq

```
seq       :: Parser a → Parser b → Parser (a,b)
p 'seq' q = \s → [((v,w), s'') | (v, s')   ← p s
                               , (w, s'')  ← q s']
```

Комбинатор последовательного применения `seq` принимает на вход два анализатора `p`, `q` и возвращает анализатор, который последовательно запускает анализатор`p` и `q`. В качестве результата `seq` возвращает список пар вида `(x, y)`, где `x` - пара из результатов разбора `p` и `q`, `y` - оставшаяся часть входного потока.

Листинг 1.2: Комбинатор bind

```
bind      :: Parser a → (a → Parser b) → Parser  b
p 'bind' f = \s → concat[f v s' | (v, s') ← p s]
```

Анализатор `p` применяется к входной строке `s` и возвращает список пар вида (значение, строка). Функция `f` берет значение и возвращает анализатор, который применяется к каждому значению по очереди. В результате `bind` вернет список пар вида (значение, строка).

Однако наивная реализация этого подхода может иметь экспоненциальную сложность разбора, еще одним недостатком является невозможность использования леворекурсивных правил.

2.2 Булевы грамматики

На практике часто встречаются языки, которые нельзя описать с помощью контекстно-свободных грамматик. Например, рассмотрим язык $\{a^m b^n c^n | m, n \geq 0, m \neq n\}$, которой не является контекстно-свободным. Его можно задать с помощью булевой грамматики:

$$
\begin{aligned}
S &\rightarrow AB \,\&\, \neg\, DC \\
A &\rightarrow aA \mid \varepsilon \\
B &\rightarrow bBc \mid \varepsilon \\
C &\rightarrow cC \mid \varepsilon \\
D &\rightarrow aDb \mid \varepsilon
\end{aligned}
$$

Рис.1: Булева грамматика для языка $\{a^m b^n c^n | m, n \geq 0, m \neq n\}$

Булева грамматика - это четверка $G = (\Sigma, N, P, S)$, где:

- Σ – конечное непустое множество терминалов;
- N – конечное непустое множество нетерминалов;
- P – конечное множество правил.

Каждое правило имеет следующий вид:

$$A \to \alpha_1 \& ... \& \alpha_m \& \neg\beta_1 \& ... \& \neg\beta_n$$

$$(m + n \geq 1, \alpha_i, \beta_i \in (\Sigma \cup N)^*)$$

Правило A можно интерпретировать следующим образом: если строка представима в форме $\alpha_1, ..., \alpha_m$, но не представима в форме $\beta_1., ..., \beta_n$, то она выводится из нетерминала A.

Булевы грамматики расширяют контекстно-свободные грамматики, позволяя использовать в правых частях правил вывода конъюнкцию и отрицание.

2.3 Continuation Passing Style

Наивная реализация парсер-комбинаторного подхода имеет несколько ограничений. Во-первых, при наивном подходе имеет значение порядок альтернатив: анализатор завершает работу, пройдя по первой успешной альтернативе. Во-вторых, анализатор возвращает лишь одно дерево вывода. Поскольку потенциально анализатор может успешно завершить работу несколько раз на одной позиции во входном потоке, для разбора неоднозначных грамматик необходим исчерпывающий поиск, т.е. поиск всех возможных выводов строки. Одним из способов достичь этого является техника программирования в стиле передачи продолжений (Continuation Passing Style). Идея техники заключается в передаче управления через механизм продолжений. Продолжение представляет собой состояние программы, в которое может быть осуществлен переход из любой точки программы.

Рассмотрим простой пример – определение функции, возводящей число в квадрат.

Листинг 1.3: Обычное определение функции

```
square :: Int → Int
square x = x*x
```

Листинг 1.4: Определение функции в стиле передачи продолжений

```
square_cps :: Int → (Int → c) → c
square_cps x k = k (x*x)
```

У каждой функции есть дополнительный аргумент – функция, которой будет передан результат. В примере выше `k` – это продолжение, которому передается результат возведения числа в квадрат. Например, в качестве продолжения может быть передана функция, выводящая результат в консоль.

2.4 Мемоизация вычислений синтаксического анализа

Использование мемоизации при построении нисходящих синтаксических анализаторов для избежания экспоненциального времени анализа впервые было предложено в работе [9]. Идея техники заключается в сохранении результата работы анализатора для предотвращения повторых вычислений: если анализатор уже вызывалась на `i`-ой позиции, возвращается результат из таблицы, в противном случае анализатор запускается на позиции `i`, а результат затем помещается в таблицу.

2.5 Поддержка левой рекурсии

При наивной реализации парсер-комбинаторного подхода вызов анализатора на грамматике вида $E \to E\ E \mid a \mid \varepsilon$ не завершится, поскольку анализатор зациклится. В связи с этим становится актуальным вопрос поддержки левой рекурсии. В данном разделе описан подход к поддержке леворекурсивных правил. Впервые он был представлен в работе [7].

Заведем функцию `memo`, которая принимает на вход анализатор `f` и возвращает его мемоизированную версию, которая каждый раз при вызове на `i`-ой позиции обращается к таблице мемоизации. Если анализатор уже вызывался на `i`-ой позиции, то возвращается соответствующее значение, в противном случае вызывается функция memoresult с аргументом `f(i)`. Благодаря стратегии call-by-name анализатор `f` не запускается на позиции `i`, что гарантирует единственность вызова. Функция memoresult возвращает объект, у которого есть доступ к двум спискам: `Rs` хранит позиции, на которых работа анализатора завершилась успешно, `Ks` хранит все продолжения, переданные немемоизированному анализатору при вызове на `i`-ой позиции. Если анализатор вызван впервые, то текущее продолжение добавляется к списку `Ks`. При вызове анализатора на позиции `j`, которой нет в `Rs`, сначала `j` добавляется в `Rs`, затем запускаются все продолжения из `Ks` на `j`-ой позиции. Если анализатор уже вызывался, то текущее продолжение добавляется к `Ks` и вызывается для каждой позиции из `Rs`.

Теперь, когда мемоизированный анализатор вызывается на `i`-ой позиции, его завершение будет гарантировано, поскольку не будет вызван на `i`-ой позиции больше одного раза.

2.6 Существующие решения

В данном разделе приведен обзор парсер-генераторной библиотеки с поддержкой булевых грамматик и двух парсер-комбинаторных библиотек.

Whale Calf Whale Calf[1] - инструмент для разбора булевых грамматик, реализованный А.Охотиным на языке `C++`. Инструмент состоит из двух компонент: парсер-генератор, который преобразует текстовое описание конъюнктивных грамматик в исполняемый код, и библиотека, которая используется

[1] `http://users.utu.fi/aleokh/whalecalf/`

для анализа грамматик. В библиотеке реализовано несколько алгоритмов синтаксического анализа: табличный алгоритм для грамматик в бинарной нормальной форме, табличный алгоритм для грамматик в линейной нормальной форме, табличный алгоритм для произвольных грамматик, конъюнктивный LL, конъюнктивный LR и алгоритм, основанный на эмуляции автоматов, эквивалентных линейным конъюнктивным грамматикам.

Рассмотрим пример описания грамматики в контексте данной библиотеки.

Листинг 1.5: Описание грамматики для языка $\{\omega c \omega | \omega \in a, b^*\}$

```
algorithm=LR;
terminal a, b, c;

S → C & D;
C → a C a | a C b | b C a | b C b | c;
D → a A & a D | b B & b D | c E;
A → a A a | a A b | b A a | b A b | c E a;
B → a B a | a B b | b B a | b B b | c E b;
E → a E | b E | e;
```

Whalf Calf обрабатывает файл с описанием грамматики и создает два файла с определением анализатора на языке C++ – `filename.cpp` и `filename.h`. Рассмотрим несколько методов, представленных в примере ниже:

- `read()` используется для передачи входной строки;
- `recognize()` используется для определения принадлежности строк языку;
- `parse()` используется для построения дерева вывода.

Листинг 1.6: Пример использования анализатора

```
int x[] = {0, 1, 2, 0, 1}; // input "abcab"
whale_calf.read(x, x + 5);
WhaleCalf::TreeNode *tree = whale_calf.parse();
if (tree) whale_calf.print_tree(ofstream("parse_tree.dot"));
```

Данная реализация имеет сложность $O(n^4)$ по времени. Утверждается, что на практике работает быстрее, чем реализации других алгоритмов, имеющих в худшем случае сложность $O(n^3)$.

Meerkat Meerkat[2] - парсер-комбинаторная библиотека с поддержкой всех контекстно-свободных грамматик, в том числе содержащих леворекурсивные правила, написанная на языке `Scala`. Библиотека позволяет строить дерево разбора за линейное время на грамматиках реальных языков программирования и за кубическое время в худшем случае. Помимо прочего, строится SPPF (Shared Packed Parse Forest) – графовое представление семейства деревьев разбора. Описание подхода к поддержке левой рекурсии описано авторами в работе [3].

В языке Scala функции и переменные могут состоять из любых символов, например, можно создать функцию с именем $\sim$. В примере выше $|$ - комбинатор альтернативного применения, $\sim$ - комбинатор последовательного применения. Для указания приоритета используется $|>$, для указания левой и правой ассоциативности используются `left` и `right` соотвественно. Терминальные символы могут быть представлены строками - “$-$”, “$+$”, “$*$” и т.д.

3 Реализация

В данном разделе описана реализация парсер-комбинаторной библиотеки.

В качестве языка для реализации выбран Kotlin[3]. Kotlin – это молодой язык программирования, поддерживающий объектно-ориентированную и функциональную парадигмы, разрабатывается компанией JetBrains[4]. Поскольку Kotlin базируется на JVM, на ранних этапах разработки библиотеки

[2] `http://meerkat-parser.github.io/`
[3] `http://kotlinlang.org/`
[4] `http://jetbrains.com/`

была актуальна проблема переполнения стека из-за большого количества функциональных вызовов. Решить проблему удалось, воспользовавшись техникой программирования в стиле передачи продолжений (Continuation Passing Style).

3.1 Интерфейс библиотеки

Анализатор имеет тип `Recognizer<A>`, принимает на вход позицию во входном потоке и возвращает функцию типа `CPSResult<A>`. Эта функция принимает на вход продолжение типа `K<A>` и возвращает `Unit`. Тип `Unit` эквивалентен типу `void` в других языках программирования. Продолжение представляет собой следующий этап синтаксического анализа. Анализ производится вне зависимости от порядка альтернатив, таким образом обеспечивается исчерпывающий поиск.

Листинг 1.7: Основные типы

```
typealias Recognizer<A> = (Int)     → CPSResult<A>
typealias CPSResult <A> = (K<A>)    → Unit
typealias K<A>          = (Int, A) → Unit
```

В библиотеке реализовано множество различных анализаторов и комбинаторов, среди которых можно выделить несколько базовых:

- `terminal` - принимает на вход строку `s` и проверяет, начинается ли входная строка с подстроки `s`;
- `eps` – анализатор для пустой строки;
- `rule` – комбинатор альтернативного применения;
- `seq` – комбинатор последовательного применения;
- `map` – принимает на вход анализатор `p` и функцию `f`, применяет `f` к результату работы `p`;
- `fix` – комбинатор неподвижной точки, используется для определения рекурсивных правил;
- `number` – анализатор чисел;
- `symbol` – анализатор последовательности букв и чисел;

– `paren` – принимает на вход анализатор`f` и проверяет, заключен ли результат `f` в круглые скобки во входной строке.

В качестве примера работы с библиотекой рассмотрим реализацию анализатора для языка While. While – простой язык программирования, в котором определены следующие конструкции: присваивание, последовательная композиция, условный оператор и while.

Определение анализатора для разбора выражений приведено в листинге 9. Оператор "/" представляет собой синтаксический сахар для комбинатора альтернативного применения.

Листинг 1.8: Анализатор выражений для языка While

```
1  val exprParser: Recognizer<Expr> = fix {
2      val corep = (number map { Expr.Con(it) as Expr }) /
3                  (symbol map { Expr.Var(it) as Expr }) /
4                  paren( sp(it) )
5      val op1p = rightAssocp(sp(terminal("^")), corep) {
6                      l, op, r → Expr.Binop(l, op, r)
7              }
8      val op2p =
9        rightAssocp(sp(terminal("*") / terminal("/") /
10         terminal("%")), op1p) {
11           op, e1, e2 →
12             Expr.Binop(op, e1, e2)
13         }
14     val op3p =
15       assocp(sp(terminal("+") / terminal("-")), op2p) {
16         op, e1, e2 →
17           Expr.Binop(op, e1, e2)
18       }
19     return@fix op3p
20 }
```

3.2 Поддержка булевых грамматик

Для поддержки булевых грамматик необходимо определить две операции – конъюнкцию и отрицание. Комбинатор

`And`, реализующий операцию конъюнкция, должен находить пересечение языков, порождаемых поступившими на вход анализаторами. Если входная строка принадлежит пересечению, то анализ завершается успешно. Комбинатор `AndNot`, реализующий операцию отрицание, завершает работу успешно, если входная строка принадлежит одному языку, а другому – нет. Ниже приведено подробное описание реализации обоих комбинаторов.

Комбинатор And принимает в качестве параметров два анализатора `p1` и `p2` и возвращает анализатор, который работает следующим образом:

- хранит два списка пар для `p1` и `p2` соответственно вида <`Int`, `A`>, где первый параметр - номер позиции во входном потоке, второй параметр - обработанный символ входного потока;
- запускает оба анализатора на `i`-ой позиции с продолжением, которое принимает пару вида <`Int`, `A`>, где первый параметр - `pos` - позиция во входном потоке, второй параметр - `res` - обработанный символ входного потока. Добавляет пару в соответствующий список пар для анализатора `p1`(`p2`). Проверяет, есть ли ли в соответствующем списке для анализатора `p2`(`p1`) пара, в которой первый элемент соответствует равен `pos`. Если да, то исходному продолжению передается пара, состоящая из `pos` и пары, которая состоит из `pos` и `res`;
- анализатор сообщает, что строка принадлежит пересечению конъюнктов, если порождается и анализатором `p1`, и анализатором `p2`.

Листинг 1.9: Код комбинатора And

```
1  class And<A, B> (
2  val p1: Recognizer<A>, val p2: Recognizer<B>
3  ) : Recognizer<Pair<A, B>>() {
4      override fun invoke(p: Int): CPSResult<Pair<A, B>> {
5          val passedByp1: ArrayList<Pair<Int, A>> = ArrayList()
```

```
6           val passedByp2: ArrayList<Pair<Int, B>> = ArrayList()
7           val executed  : ArrayList<Pair<Int, Pair<A, B>>> =
8             ArrayList()
9
10          return { k: K<Pair<A, B>> →
11              p1(p)({ pos: Int, res: A →
12                  val pair1p = Pair(pos, res)
13                  passedByp1.add(pair1p)
14                  passedByp2.forEach { r →
15                      if (pair1p.first == r.first) {
16                          val kpair =
17                            Pair(pair1p.second, r.second)
18                          k(pos, kpair)
19                          executed.add(Pair(pos, kpair))
20                      }
21                  }
22              })
23
24              p2(p)({ pos: Int, res: B →
25                  val pair2p = Pair(pos, res)
26                  passedByp2.add(pair2p)
27                  passedByp1.forEach { r →
28                      if (pair2p.first == r.first) {
29                          val kpair =
30                            Pair(r.second, pair2p.second)
31                          k(pos, kpair)
32                          executed.add(Pair(pos, kpair))
33                      }
34                  }
35              })
36          }
37      }
38  }
```

Строка выводится по правилу $S \to A\&\neg B$, если она может быть порождена из нетерминала A и не может быть порождена из нетерминала B. Таким образом, нужен комбинатор `AndNot`, проверяющий это условие. В качестве параметров комбинатор

`AndNot` принимает два анализатора `p1` и `p2` и возвращает анализатор, который работает следующим образом:

- хранит два списка пар для `p1` и `p2` соответственно вида `<Int, A>`, где первый параметр - номер позиции во входном потоке, второй параметр - обработанный символ входного потока;
- запускает анализатор `p1` на `i`-ой позиции с продолжением, которое добавляет пару типа `<Int, A>` в соответствующий список для `p1`. Эта пара передается продолжению `k`, если анализ прошел успешно;
- затем запускает анализатор `p2` на `i`-ой позиции;
- если анализатор `p2` не разобрал входную строку до той же позиции, что и анализатор `p1`, то анализ входной строки завершается успешно.

Листинг 1.10: Код комбинатора AndNot

```
1  class AndNot<A, B> (
2  val p1: Recognizer<A>, val p2: Recognizer<B>
3  ) : Recognizer<A>() {
4      override fun invoke(p: Int): CPSResult<A> {
5          val passedByp1: ArrayList<Pair<Int, A>> = ArrayList()
6          val passedByp2: ArrayList<Pair<Int, B>> = ArrayList()
7          val executed  : ArrayList<Pair<Int, A>> = ArrayList()
8          return { k: K<A> →
9              p2(p)({ pos: Int, res: B →
10                 val pair2p = Pair(pos, res)
11                 passedByp2.add(pair2p)
12                 passedByp1.forEach { r →
13                     val q = passedByp2.find {it.first == r.first}
14                     if(q == null){
15                         k(pos, r.second)
16                         executed.add(Pair(pos, r.second))
17                     }
18                 }
19             })
20             p1(p)({ pos: Int, res: A →
```

```
21                  val pair1p = Pair(pos, res)
22                  passedByp1.add(pair1p)
23                   k(pos, res)
24              })
25          }}}
```

4 Апробация

В данном разделе приведены результаты тестирования производительности парсер-комбинаторной библиотеки. Для тестирования были выбраны несколько грамматик. Эксперименты проводилось на машине со следующими характеристиками:

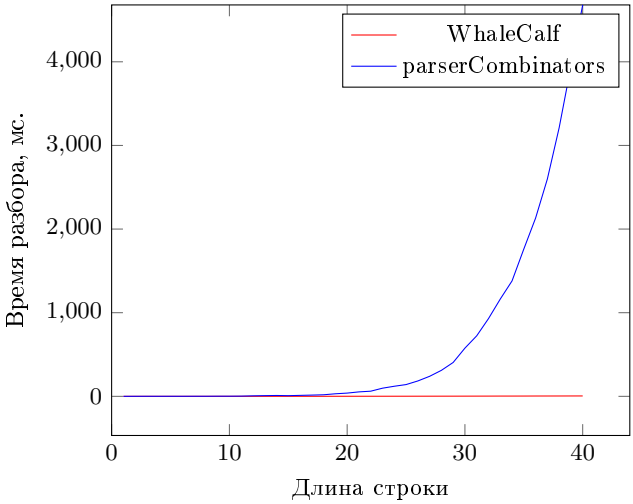
- OS – Ubuntu 16.04;
- CPU – Intel Core i5-4200M;
- RAM – 4GB.

Первая грамматика сильно неоднозначная, содержащая леворекурсивные правила, реализует худший случай для анализатора.

Листинг 1.11: Грамматика для языка $\{a^*\}$

```
S → SSS | SS | a
```

Рис.2: Сравнение времени работы



Результаты измерений представлены на рис.2. В библиотеке WhaleCalf реализовано несколько алгоритмов синтаксического анализа. Лучший результат демонстрирует алгоритм GLR, поэтому для сравнения производительности использовался он. Видно, что время работы алгоритма GLR растет значительно медленнее, чем время работы парсер-комбинаторов, с ростом длины входной строки.

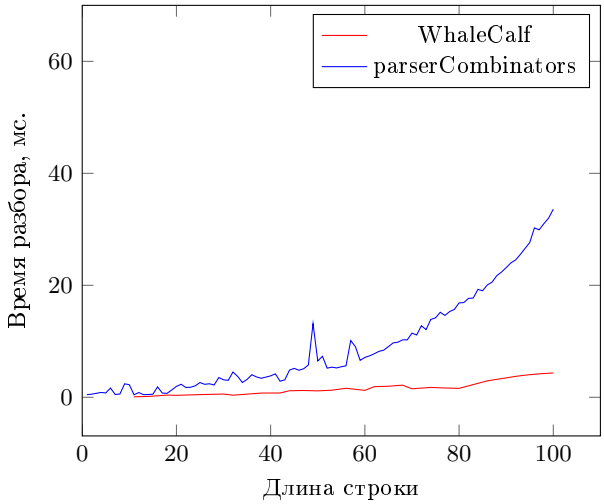
Следующий эксперимент проводился на конъюнктивной грамматике, которая определяет требование declaration before use. Грамматика взята из работы [2]. Результаты измерений представлены на рис.3. Видно, что решение, основанное на алгоритме GLR, быстрее парсер-комбинаторов.

Листинг 1.12: Грамматика для языка $\{u_1 ... u_n | n \geq 0, \forall i \ u_i \in da^* \ or \ u_i = ca^k \ and \ u_j = da^k \ \forall j < i, \ k \geq 0\}$

```
S → SdA | ScA & EdB | ε
A → aA  | ε
```

```
B → aBa | E c
E → EcA | EdA | ε
```

Рис.3: Сравнение времени работы

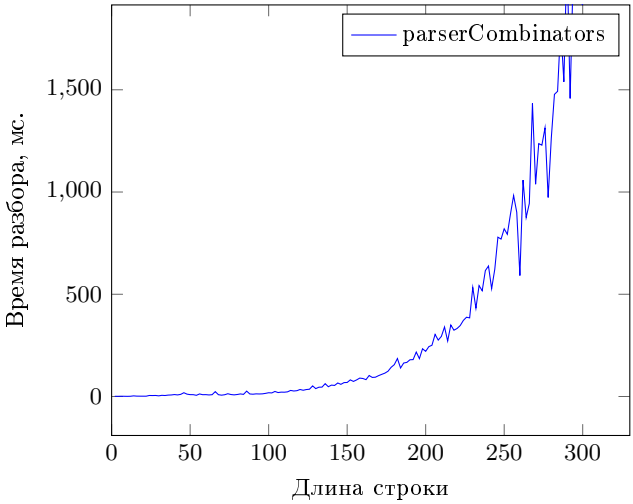


Также тестирование проводилось на булевой грамматике для языка $\{\omega\omega|\omega \in \{a,b\}^*\}$. Результаты представлены на Рис. 4.

Листинг 1.13: Булева рамматика для языка $\{\omega\omega|\omega \in \{a,b\}^*\}$

```
S → ¬AB & ¬BA & C
A → XAX | a
B → XBX | b
X → a   | b
C → XXC | ε
```

Рис.4: Время работы



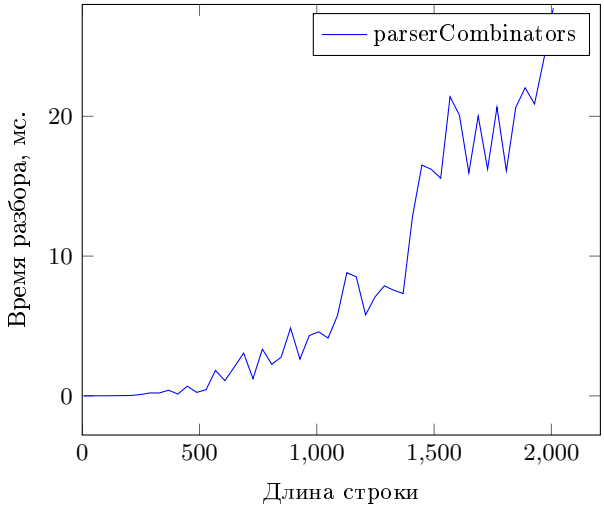
Базовая структура псевдоузла (pseudoknot), элемента вторичной структуры РНК, может быть представлена в виде языка $L_2 = \{\omega|\omega = [^i\ (^j\ ]^i\ )^j\}$, где пара скобок представляет собой пару оснований (base pair). Язык L_2 не является контекстно-свободным, однако он может быть представлен в виде пересечения языков $\{\omega|\omega = [^i\ (^*\ ]^i\ )^*\}$ и $\{\omega|\omega = [^*\ (^j\ ]^*\ )^j\}$.

Листинг 1.14: Конъюнктивная рамматика для языка L_2

```
S  → S1 & S5
p1 → [
p2 → ]
p3 → (
p4 → )
S1 → S2 S3
S2 → p1 S2 p2 | S4
S4 → p3 S4 |  ε
S3 → p4 S3 |  ε
S5 → S6 S7
```

```
S7 → p3 S7 p4 | S8
S6 → p1 S6 | ε
S8 → p2 S8 | ε
```

Рис.5: Время работы



Результаты измерений представлены на Рис.5. Видно, что время работы растет достаточно быстро.

Заключение

В ходе работы получены следующие результаты:

- реализована парсер-комбинаторная библиотека с поддержкой булевых грамматик, в том числе содержащих леворекурсивные правила;
- проведена апробация библиотеки на примере контекстно-свободной грамматики, булевой грамматики и грамматики из биоинформатики;
- проведено сравнение с существующими решениями;

– результаты работы представлены на конференции "СПИСОК-2017".

Исходный код доступен в репозитории [10], автор вел работу под учетной записью *onewhl*.

В текущем виде библиотека не готова к промышленному использованию, поскольку необходимы оптимизации, способствующие повышению скорости работы парсер-комбинаторов.

Список литературы

1. Alexander Okhotin. Boolean grammars // Information and Computation. — 2004. — Vol. 194, no. 1. — P. 19–48.
2. Alexander Okhotin. Conjunctive and Boolean grammars: the true general case of the context-free grammars // Computer Science Review. — 2013. — Vol. 9. — P. 27–59.
3. Anastasia Izmaylova, Ali Afroozeh, Tijs van der Storm. Practical, general parser combinators // Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation / ACM. — 2016. — P. 1–12.
4. Daniel Spiewak. Generalized Parser Combinators. — 2010.
5. Graham Hutton, Erik Meijer. Monadic parser combinators. — 1996.
6. Manfred J Sippl. Biological sequence analysis. Probabilistic models of proteins and nucleic acids, edited by R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson. 1998. // Protein Science. — 1999. — Vol. 8, no. 3. — P. 695–695.
7. Mark Johnson. Memoization in top-down parsing // Computational Linguistics. — 1995. — Vol. 21, no. 3. — P. 405–417.
8. Noam Chomsky. Three models for the description of language // IRE Transactions on information theory. — 1956. — Vol. 2, no. 3. — P. 113–124.
9. Peter Norvig. Techniques for automatic memoization with applications to context-free parsing // Computational Linguistics. — 1991. — Vol. 17, no. 1. — P. 91–98.
10. parserCombinators [Электронный ресурс]. — Режим доступа: `https://github.com/anlun/parserCombinators/` (дата обращения: 19.05.2017).

Поддержка расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalised LL

Горохов Артем Владимирович

Санкт-Петербургский государственный университет
gorohov.art@gmail.com

Аннотация Для автоматизации разработки синтаксических анализаторов часто используют генераторы, строящие анализаторы на основе спецификации языка. Обычно спецификация описывается неоднозначной грамматикой в расширенной форме Бэкуса-Наура (EBNF), которую не могут обрабатывать большинство инструментов без преобразования к форме Бэкуса-Наура (BNF). Существующие подходы анализа EBNF без преобразования к BNF не способны обрабатывать неоднозначные грамматики. С другой стороны, алгоритм Generalized LL (GLL) допускает произвольные контекстно-свободные грамматики и достигает высокой производительности, но не способен обрабатывать EBNF грамматики. В данной работе описана модификация GLL алгоритма для обработки расширенных контекстно-свободных грамматик, эта форма родственна EBNF. Так же показано, что предложенный подход повышает производительность анализа по сравнению с алгоритмами использующими преобразование грамматик.

Введение

Статический анализ — это анализ программного кода без его исполнения. Он производится после синтаксического анализа на основе грамматики, описывающей синтаксис языка. Общеупотребимый способ описания синтаксиса языков программирования — грамматики в расширенной форме Бэкуса-Наура (EBNF) [29]. С одной стороны, эта форма проста для понимания людей, с другой, достаточно формальна и допускает автоматизированное создание синтаксических анализаторов. Примером могут служить спецификации языков C, $C++$, *Java* и т.д.

Проблема в том, что существующие генераторы анализаторов, такие как ANTLR [1], Bison [5], преобразуют грамматики в форму EBNF для упрощения их структуры. В результате этого, представление кода строится на основе грамматики, отличной от заданной, что затрудняет обработку результатов анализа. С другой стороны, производительность таких алгоритмов анализа, как Generalised LL (GLL) [22] зависит от структуры грамматики, а её упрощение часто ведёт к избыточности представления, что отрицательно сказывается на производительности.

Алгоритм GLL показывает высокую производительность и способен работать с неоднозначными грамматиками. Предполагается, что поддержка в нём EBNF или родственных этой форме расширенных контекстно-свободных грамматик, увеличит производительность анализа для некоторых грамматик.

В данной работе предложен алгоритм синтаксического анализа, основанный на Generalised LL, для работы с расширенными контекстно-свободными грамматиками без преобразований. Показанно, что для некоторых грамматик алгоритм более производителен, чем существующие вариации GLL. В разделе 1 представлен обзор предметной области и литературы. Далее в разделе 2 представлены необходимые изменения в Generalised LL алгоритме и сопутствующих структурах данных. В разделе 3 описаны пути применения описанного ал-

горитма в задаче анализа регулярных множеств. Эксперементальное сравнение полученного алгоритма с современными модификациями Generalised LL содержится в разделе 4.

1 Обзор предметной области

Заметим, что EBNF является стандартизированной формой для *расширенных контекстно-свободных грамматик.*

Определение 1. *Расширенная контекстно-свободная грамматика (ECFG) [12] — это кортеж* (N, Σ, P, S)*, где* N *и* Σ *конечные множества нетерминалов и терминалов соответственно,* $S \in N$ *является стартовым символом, а* P *(продукция) является отображением из* N *в регулярное выражение над алфавитом* $N \cup \Sigma$*.*

Существует широкий спектр методов анализа и алгоритмов [4, 7, 9–12, 15, 17], предназначенных для обработки ECFG. Детальный обзор результатов и задач в этой области представлены в работе [12]. Следует отметить, что большинство алгоритмов основано на методах LL-анализа [4, 8, 10] и LR-анализа [7, 15, 17], но они работают только с ограниченными подклассами расширенных контекстно-свободных грамматик — LL(k), LR(k). Таким образом, нет решения для обработки произвольных (в том числе неоднозначных) ECFG-грамматик.

ECFG-грамматики широко используется в качестве входного формата для генераторов синтаксических анализаторов, но классические алгоритмы синтаксического анализа часто требуют форму Бэкуса-Наура (BNF), в продукциях которой допускаются лишь последовательности из терминалов и нетерминалов. Возможно преобразование грамматик ECFG в форму BNF [11], но оно приводит к увеличению размера грамматики и изменению её структуры: при трансформации добавляются новые нетерминалы. В результате синтаксический анализатор строит дерево вывода относительно преобразованной грамматики, что затрудняет обработку результата анализа.

В настоящее время алгоритмы на основе LL(1)-анализа представляются наиболее практичными и обеспечивают лучшую диагностику ошибок по сравнению с LR-анализом, так как являются низходящими. Но некоторые грамматики не являются LL(k) (для любого k) и не могут быть использованы в LL(k) анализаторах. Другие проблемы для инструментов на основе LL — леворекурсивные грамматики и неоднозначности в грамматике, которые, вместе с предыдущим недостатком, усложняют создание анализаторов. Алгоритм Generalised LL, предложенный в [22], решает все эти проблемы: он обрабатывает произвольные CFG, в том числе неоднозначные и леворекурсивные. В общем случае временная и пространственная сложность GLL зависит кубически от размера входа. А для LL(1) грамматик, он демонстрирует линейную временную и пространственную сложность. Но он, как и другие современные алгоритмы, не допускает использования ECFG без предварительного преобразования к форме BNF.

Для увеличения производительности Generalised LL-алгоритма, была предложена поддержка лево-факторизованных грамматик в нём [24]. Алгоритм GLL обрабатывает все возможные ветви разбора строки по заданной грамматике, эти ветви описываются так называемыми дескрипторами, которые состоят из позиций в грамматике и во входе, корня построенного леса разбора и текущего узла стека. Из этого следует, что для уменьшения времени анализа и количества используемой памяти можно снизить количество дескрипторов для обработки. Один из путей для достижения этого — уменьшение размера грамматики (снижение количества различных позиций в ней). Этого можно достичь факторизацией грамматики. Пример факторизации показан на рис. 1: из грамматики G_0 в процессе факторизации получена грамматика G'_0. Этот пример рассмотрен в работе [24], и доказано, что для некоторых грамматик факторизация существенно увеличивает производительность алгоритма GLL.

$S ::= a\ a\ b\ c\ d \mid a\ a\ c\ d \mid a\ a\ c\ e \mid a\ a$

(a) Исходная грамматика G_0

$S ::= a\ a\ (b\ c\ d \mid c\ (d \mid e) \mid \varepsilon\ )$

(b) Факторизованная грамматика G'_0

Рис. 1: Пример факторизации грамматики

Одно из возможных применений обобщённого синтаксического анализа — синтаксический анализ регулярных множеств. Синтаксическим анализом регулярных множеств называют анализ строк, заданных всеми возможными путями в конечном автомате. Такая задача возникает в различных областях одна из которых — биоинформатика. В результате экспериментов получают данные о геномах организмов, которые представлены строками в конечном автомате (геномы это строки над алфавитом $\{A; C; G; T\}$). Этот автомат называется метагеномной сборкой.

Существует множество подходов к анализу и идентификации геномов. Один из них — поиск и сравнение участков таких структур как 16s рРНК, тРНК, так как по ним можно достаточно точно классифицировать организм, которому они принадлежат. Существуют такие инструменты, как READO [20] и HMMER [28], использующие скрытые цепи Маркова для поиска, Infernal [16], использующий ковариационные модели. Но они работают лишь с линейными цепочками генома — не объединёнными в конечный автомат, такое представление требует большого количества памяти и неэффективно. С другой стороны, инструмент Xander [30] использует композицию скрытых моделей Маркова и метагеномных сборок. Изъян данного инструмента в существенно более низкой точности результата в сравнении с инструментами, использующими ковариационные модели.

Другой подход разрабатывается в лаборатории JetBrains СПбГУ. Поиск производится непосредственно в метагеномной сборке по таким структурам как тРНК, 16s рРНК. Эти

структуры имеют некоторые общие свойства в строении, которые могут быть описаны контекстно-свободной грамматикой [8]. Таким образом, можно использовать синтаксический анализ регулярных множеств для поиска. Этот подход описан в работе [19], основан на алгоритме синтаксического анализа Generalised LL и был реализован в рамках проекта YaccConstructor [31]. В нашей работе будут использоваться результаты и данные из работы [19] для сравнения производительности полученного алгоритма.

2 Использование Generalised LL для обработки ECFG

В этом разделе описываются структуры и методы необходимые для использования расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalised LL, а так же необходимые изменения в алгоритме анализа.

2.1 Представление ECFG

Представление ECFG, наиболее подходящее для синтаксического анализа — рекурсивные автоматы (Recursive Automaton (RA) [26].

Определение 2. *Рекурсивный автомат R это кортеж $(\Sigma, Q, S, F, \delta)$, где Σ — конечное множество терминалов, Q - конечное множество состояний, $S \in Q$ — начальное состояние, $F \subseteq Q$ — множество конечных состояний, $\delta : Q \times (\Sigma \cup Q) \to Q$ — функция перехода.*

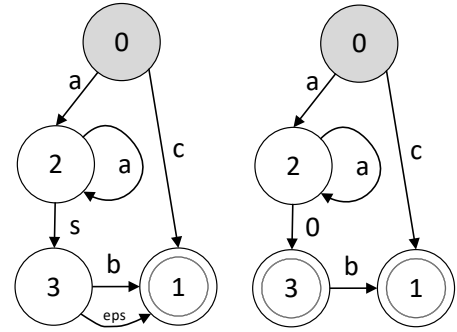
В рамках этой работы единственное различие между рекурсивным автоматом и общеизвестным конечным автоматом (FSA) состоит в том, что переходы в RA обозначаются либо терминалом (Σ), либо состоянием автомата (Q). Далее будем называть переходы помеченные элементами из Q *нетерминальными переходами*, а терминалами — *терминальными*

переходами. Нетерминальный переход в состояние q подразумевают построение вывода для некоторой подстроки начиная с текущей позиции во входе по этому нетерминалу и последующий разбор оставшейся подстроки начиная с состояния q.

Заметим, что позиции грамматики эквивалентны состояниям автомата, которые строятся из правых частей продукций грамматики. Правые части продукций ECFG являются регулярными выражениями над объединенным алфавитом терминалов и нетерминалов, поэтому построить по ним автомат можно использую общеизвестные алгоритмы. Следующий алгоритма строит RA с минимальным числом состояний для заданной ECFG.

- Построить конечный автомат, используя метод Томпсона [27] для правых частей продукций.
- Создать ассоциативный массив M из каждого нетерминала в соответствующее начальное состояние автомата. Этот массив должен оставаться консистентным на протяжение всех следующих шагов.
- Преобразовать автоматы из предыдущего шага в детерминированные без ε-переходов используя алгоритм, описанный в [3].
- Минимизировать детерминированный автомат, используя, например, алгоритм Джона Хопкрофта [13].
- Заменить нетерминальные переходы переходами по, стартовым состояниям автоматов, соответствующим данным нетерминалам, используя массив M. Результат этого шага — искомый рекурсивный автомат. Также используем M для определения функции $\Delta : Q \to N$ где N — имя нетерминала.

Пример преобразования ECFG в RA представлен на рис. 2, где состояние 0 — начальное состояние полученного RA.



$S ::= a^+ S\ b? \mid c$

(a) Грамматика G_1 (b) Конечный автомат для G_1 (c) Рекурсивный автомат R_1 для G_1

Рис. 2: Преобразование грамматики в рекурсивный автомат

2.2 Лес разбора для ECFG

Результатом синтаксического анализа является структурное представление входа — дерево или лес разбора в случае нескольких вариантов деревьев. Для начала, определим дерево вывода для рекурсивного автомата: это дерево, корень которого помечен начальным состоянием, листья терминалы или ε, а внутренние узлы нетерминалы N и их дети образуют последовательность заданную путём в автомате, который начинается в состоянии q_i, где $\Delta(q_i) = N$. Введём это определение более формально.

Определение 3. *Дерево вывода последовательности α для рекурсивного автомата $R = (\Sigma, Q, S, F, \delta)$ это дерево со следующими свойствами:*

- *корень помечен $\Delta(S)$;*
- *листья — терминалы $a \in (\Sigma \cup \varepsilon)$;*
- *остальные узлы — нетерминалы $A \in \Delta(Q)$;*
- *у узла с меткой $N_i = \Delta(q_i)$ существует:*

- *дети* $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ *тогда и только тогда, когда существует путь* p *в* R, $p = q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, *где* $q_m \in F$, $l_i = \begin{cases} k_i, \ \textit{if } k_i \in \Sigma, \\ \Delta(k_i), \ \textit{if } k_i \in Q, \end{cases}$
- *только один ребенок помеченный* ε *тогда и только тогда, когда* $q_i \in F$.

Для произвольных грамматик RA может быть неоднозначным с точки зрения допустимых путей, поэтому можно получить несколько деревьев разбора для одной входной строки. Shared Packed Parse Forest (SPPF) [21] может использоваться как компактное представление всех возможных деревьев разбора. Будем использовать бинаризованную версию SPPF, предложенную в [25], для уменьшения потребления памяти и достижения кубической наихудшей временной и пространственной сложности. Бинаризованный SPPF может использоваться в GLL [23] и содержит следующие типы узлов (здесь i и j — начало и конец выведенной подстроки во входной строке):

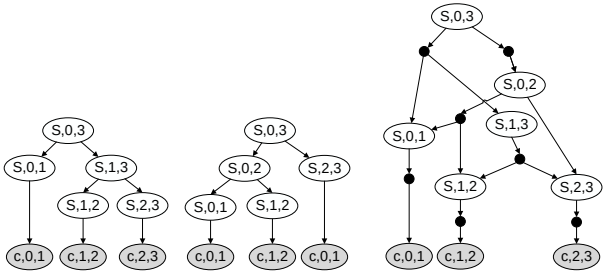
– упакованные узлы вида (S, k), где S состояние автомата, k — начало выведенной подстроки правого ребёнка; у упакованных узлов обязательно существует правый ребёнок — символьный узел, и опциональный левый — символьный или промежуточный узел;
– символьный узел помечен (X, i, j) где $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$; терминальные символьные узлы $(X \in \Sigma \cup \{\varepsilon\})$ — листья; нетерминальные символьные узлы $(X \in \Delta(Q))$ могут иметь несколько упакованных детей;
– промежуточные узлы помечены (S, i, j), где S состояние в автомате, могут иметь несколько упакованных детей.

Дети символьных и промежуточных узлов — упакованные. Различные упакованные дети — различные варианты поддеревьев. Если у узла или его потомков более одного упакованного ребёнка, то он содержит несколько вариантов разбора

для подстроки. Промежуточные и упакованные узлы необходимы для бинаризации SPPF, что обеспечивает большее переиспользование узлов. Так, деревья, представленные на рис. 4a, объединяются в SPPF показанный на рис. 4b. Опишем мо-

$$S ::= S\ S \mid c$$

Рис. 3: Грамматика G_0



(a) Возможные деревья вывода (b) SPPF

Рис. 4: Пример для входа *ccc* и грамматики G_0

дификации исходных функций построения SPPF. Функция **getNodeT**(x, i), которая создает терминальные узлы, повторно используется без каких-либо модификаций из базового алгоритма. Чтобы обрабатывать недетерминизм в состояниях, определим функцию **getNodes**, которая проверяет, является ли следующее состояние RA финальным и в этом случае строит нетерминальный узел в дополнение к промежуточному. Она использует изменённую функцию **getNodeP**: вместо позиции в грамматики он принимает в качестве входных дан-

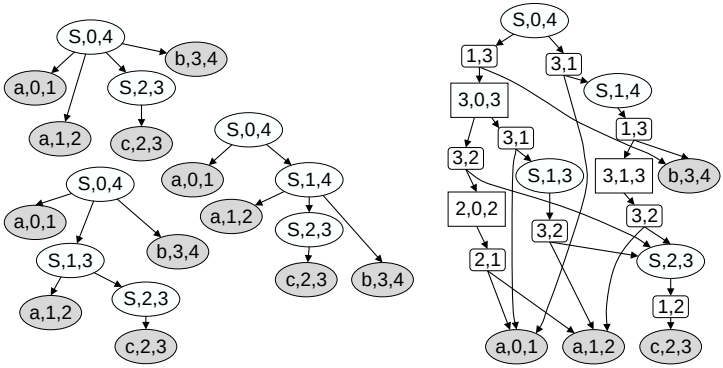
ных отдельно состояние RA и символ для нового узла SPPF: текущий нетерминал или следующее состояние RA.

```
function getNodes(S, A, w, z)
  if (S is final state) then
    x ← getNodeP(S, A, w, z)
  else x ← $
  if (w = $)& not (z is nonterminal node and its extents are
equal) then
    y ← z
  else y ← getNodeP(S, S, w, z)
  return (y, x)

function getNodeP(S, L, w, z)
  (_, k, i) ← z
  if (w ≠ $) then
    (_, j, k) ← w
    y ← find or create SPPF node labelled (L, j, i)
    if (∄ child of y labelled (S, k)) then
      y′ ← new packedNode(S, k)
      y′.addLeftChild(w)
      y′.addRightChild(z)
      y.addChild(y′)
  else
    y ← find or create SPPF node labelled (L, k, i)
    if (∄ child of y labelled (S, k)) then
      y′ ← new packedNode(S, k)
      y′.addRightChild(z)
      y.addChild(y′)
  return y
```

Рассмотрим пример SPPF для ECFG G_1, показанные на рис. 2a. Эта грамматика содержит конструкции (условное вхождение (?) и повторение (+)), которые должны быть преобразованы с использованием дополнительных нетерминалов для создания обычного GLL-анализатора. Предложенный генератор строит рекурсивный автомат R_1 (рис. 2c) и анализа-

тор для него. Возможные деревья ввода последовательности *aacb* показаны на рис. 5a. SPPF, созданный синтаксическим анализатором (рис. 5b), содержит в себе все три дерева.



(a) Возможные деревья вывода (b) SPPF

Рис. 5: Пример для входа *aacb* и автомата R_1

2.3 Алгоритм построения леса разбора для ECFG

В этом разделе описываются изменения в управляющих функциях базового алгоритма Generalised LL, необходимые для обработки ECFG. Основной цикл представленного алгоритма аналогичен базовому GLL: на каждом шаге основная функция **parse** извлекает из очереди так называемый дескриптор R — кортеж описывающий текущую ветку разбора. Пусть текущий дескриптор (C_S, C_U, i, C_N), где C_S — состояние RA, C_U — узел GSS, i — позицию во входной строке ω, C_N — узел SPPF. В ходе обработки дескриптора могут возникнуть следующие не исключающие друг друга ситуации.

- C_S — финальное состояние. Это возможно только если C_S — стартовое состояние текущего нетерминала. Следует построить нетерминальный узел с ребёнком (ε, i, i) и вызвать функцию **pop**, так как разбор нетерминала окончен.
- Существует терминальный переход $C_S \xrightarrow{\omega.[i]} q$. Во-первых, построить терминальный узел $t = (\omega.[i], i, i+1)$, далее вызвать функцию **getNodes** чтобы построить родителя для C_N и t. Функция **getNodes** возвращает кортеж (y, N), где N — опциональный нетерминальный узел. Создать дескриптор $(q, C_U, i+1, y)$ и, если в q ведёт несколько переходов, вызвать функцию **add** для этого дескриптора. Иначе поместить его в очередь вне зависимости от того был ли он создан до этого. Если $N \neq \$$, вызвать функцию **pop** для этого узла, состояния q и позиции во входе $i+1$.
- Существуют нетерминальные переходы из C_S. Это значит, что следует начать разбор нового нетерминала, поэтому должен быть создан новый узел GSS, если такового ещё нет. Для этого нужно вызвать функцию **create** для каждого такого перехода. Она осуществляет необходимые операции с GSS и проверяет наличие узла GSS для текущих нетерминала и позиции во входе.

Псевдокод для необходимых функций представлен ниже.

Функция **add** помещает в очередь дескриптор, если он не был создан до этого; эта функция не изменилась.

```
function CREATE(S_call, S_next, u, i, w)
  A ← Δ(S_call)
  if (∃ GSS node labeled (A, i)) then
    v ← GSS node labeled (A, i)
    if (there is no GSS edge from v to u labeled (S_next, w))
then
      add GSS edge from v to u labeled (S_next, w)
      for ((v, z) ∈ P) do
        (y, N) ← getNodes(S_next, u.nonterm, w, z)
        (_, _, h) ← y
        add(S_next, u, h, y)
```

```
            if N ≠ $ then
              (_, _, h) ← N; pop(u, h, N)
        else
          v ← new GSS node labeled (A, i)
          create GSS edge from v to u labeled (S_next, w)
          add(S_call, v, i, $)
        return v
    function POP(u, i, z)
        if ((u, z) ∉ P) then
          P.add(u, z)
          for all GSS edges (u, S, w, v) do
            (y, N) ← getNodes(S, v.nonterm, w, z)
            add(S, v, i, y)
            if N ≠ $ then pop(v, i, N)
    function PARSE
        GSSroot ← newGSSnode(StartNonterminal, 0)
        R.enqueue(StartState, GSSroot, 0, $)
        while R ≠ ∅ do
          (C_S, C_U, i, C_N) ← R.dequeue()
          if (C_N = $) and (C_S is final state) then
            eps ← getNodeT(ε, i)
            (_, N) ← getNodes(C_S, C_U.nonterm, $, eps)
            pop(C_U, i, N)
          for each transition(C_S, label, S_next) do
            switch label do
              case Terminal(x) where (x = input[i])
                T ← getNodeT(x, i)
                (y, N) ← getNodes(S_next, C_U.nonterm, C_N, T)
                if N ≠ $ then pop(C_U, i + 1, N)
                if S_next has multiple ingoing transitions then
                  add(S_next, C_U, i + 1, y)
                else
                  R.enqueue(S_next, C_U, i + 1, y)
              case Nonterminal(S_call)
```

create$(S_{call}, S_{next}, C_U, i, C_N)$
if SPPF node $(StartNonterminal, 0, input.length)$ exists **then**
return this node
else report failure

3 Синтаксический анализ регулярных множеств

Описанный в данной работе алгоритм можно применять для анализа регулярных множеств. При работе с конечным автоматом в качестве входных данных необходимо обрабатывать все переходы из текущей позиции (состояния) в автомате. Так, основная функция приобретает следующий вид:

function parseRegularSet
$GSSroot \leftarrow newGSSnode(StartNonterminal, StartState)$
$R.enqueue(StartState, GSSroot, StartState, \$)$
while $R \neq \varnothing$ **do**
$(C_S, C_U, i, C_N) \leftarrow R.dequeue()$
if $(C_N = \$)$ and (C_S is final state) **then**
$eps \leftarrow$ **getNodeT**(ε, i)
$(\_, N) \leftarrow$ **getNodes**$(C_S, C_U.nonterm, \$, eps)$
pop(C_U, i, N)
for each $transition(C_S, label, S_{next})$ **do**
switch *label* **do**
case $Terminal(x)$
for each (input[i] $\xrightarrow{x}$ input[k]) **do**
$T \leftarrow$ **getNodeT**(x, i)
$(y, N) \leftarrow$ **getNodes**$(S_{next}, C_U.nonterm, C_N, T)$
if $N \neq \$$ **then** **pop**(C_U, k, N)
add(S_{next}, C_U, k, y)
case $Nonterminal(S_{call})$
create$(S_{call}, S_{next}, C_U, i, C_N)$

if SPPF node $(StartNonterminal, StartState, \_)$ exists **then**
    return this node
else report failure

Позициями во входе для автомата становятся номера состояний и обрабатываются все исходящие переходы во входе. Кроме того, Функция **add** вызывается при обработке терминального перехода всегда, чтобы поддержать возможные циклы во входе. Например, для грамматики $S ::= a*$ и входного автомата на рис. 6 дескриптор будет создаваться бесконечно, если не добавить его в множество созданных, и алгоритм не остановится. Данное изменение не меняет теоретическую сложность алгоритма, но может сказаться на производительности в худшую сторону. Поэтому этот подход можно применять лишь только в случае присутствия циклов во ходе, иначе вызывать функцию **add** только при наличии нескольких входящих переходов в текущее состояние.

Рис. 6: Пример входа для грамматики $S ::= a^*$.

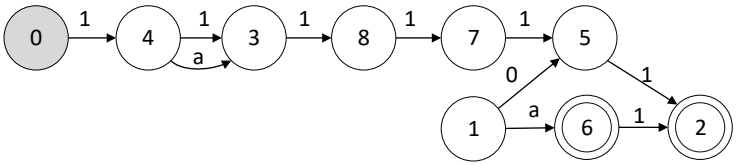
4 Эксперименты

В работе [24] было проведено экспериментальное сравнение алгоритма для факторизованных грамматик (Factorised GLL, FGLL) и базового GLL-алгоритма, продемонстрировавшее, что FGLL показывает большую производительность для грамматик в форме Бэкуса-Наура, которые могут быть факторизованы. Предполагается, что предложенная в данной работе

версия алгоритма продемонстрирует большую производительность, чем FGLL, для грамматик, имеющих эквивалентные позиции для алгоритма минимизации автомата, но различные после факторизации. Автомат, построенный для грамматики, в которой есть эквивалентные позиции, для которых алгоритм создаёт большое количество дескрипторов, объединит данные позиции, сократив тем самым множество создаваемых дескрипторов, что в свою очередь увеличит производительность. Примером такой ситуации может служить грамматика G_2 (рис. 7a), так как она содержит длинные последовательности в альтернативах, которые не сливаются при факторизации, но эквивалентны для алгоритма минимизации автомата. Рекурсивный автомат построенный для этой грамматики показан на рис. 7b.

$$S ::= K\ (K\ K\ K\ K\ K \mid a\ K\ K\ K\ K)$$
$$K ::= S\ K \mid a\ K \mid a$$

(a) Грамматика G_2



(b) Рекурсивный автомат для грамматики G_2

Рис. 7: Грамматика G_2 и RA для неё

Эксперименты проводились на входах различной длины, результаты приведены на рис. 8. Точные данные для входа a^{450} показаны в таблице 1.

Для экспериментов использовался ПК со следующими характеристиками: Microsoft Windows 10 Pro x64, Intel(R)

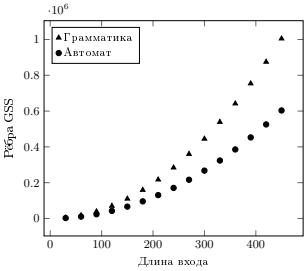
Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Cores, 4 Logical Processors, 16 GB.

	Время	Дескрипторы	Рёбра GSS	Узлы GSS	Узлы SPPF	Память, Мб
FGLL	10 мин. 13 с.	1104116	1004882	902	195 млн.	11818
RA	5 мин. 51 с.	803281	603472	902	120 млн.	8026
Ratio	43%	28%	40 %	0 %	39 %	33 %

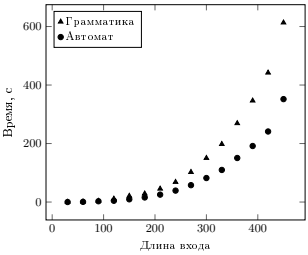
Таблица 1: Результаты экспериментов для входа a^{450}

Результаты данных экспериментов поддерживают предположение о том, что на некоторых грамматиках предложенный подход показывает результаты лучше FGLL. Для этого рекурсивного автомата анализатор создаёт меньше дескрипторов, чем для грамматики, так как цепочки нетерминалов K в альтернативах представлены единственным путём в автомате. Эта особенность ведёт к снижению количества узлов SPPF и размера GSS. В среднем, с грамматикой G_2 версия с минимизированными автоматами работает на 43% быстрее, использует на 28% меньше дескрипторов, на 40% меньше рёбер GSS, создаёт на 39% меньше узлов SPPF и использует на 33% меньше памяти.

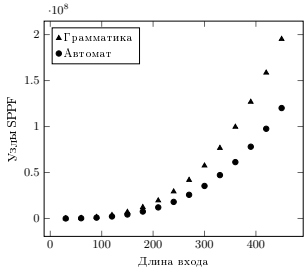
Было проведено экспериментальное сравнение разработанного алгоритма GLL с существующим в проекте YaccConstructor (основан на оригинальном алгоритме Generalised LL) в задаче поиска 16s рРНК в метагеномной сборке. Длинные рёбра сборки были предварительно отфильтрованы с помощью инструмента Infernal. В результате фильтрации сборка разбивается на компоненты, которые можно



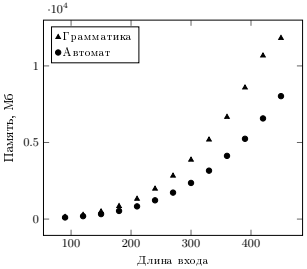
(a) Количество рёбер GSS.



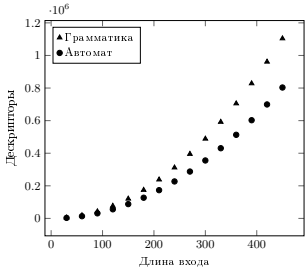
(b) Время работы.



(c) Количество узлов SPPF.



(d) Использование памяти



(e) Количество дескрипторов.

Рис. 8: Результаты экспериментов с грамматикой G_2.

анализировать независимо друг от друга. Тем не менее, предложенный ранее алгоритм не позволяет обработать некоторые компоненты, поэтому сравнение проводилось на остальных: 10 компонент с 400-100 состояний и переходов и 1118 компонент с менее чем 100 состояний и переходов. Результаты сравнения приведены в таблице 2 и показывают, что при работе с метагеномными сборками новый алгоритм, в среднем, использует на 65% меньше памяти и работает на 45% быстрее базового GLL. Сравнение с FGLL показывает на 4% меньшее использование памяти новым алгоритмом и на 10% меньшее время работы.

		GSS			
	Деск-ры	Рёбра	Узлы	Память	Время
GLL	802млн	414млн	339млн	20Гб	52 мин. 43 с.
FGLL	382млн	187млн	134млн	7Гб	29 мин. 27 с.
RA	362млн	190млн	134млн	6,8Гб	26 мин. 34 с.

Таблица 2: Результаты экспериментов с метагеномной сборкой

Заключение

В рамках данной работы была разработана и реализована модификация алгоритма GLL, работающая с расширенными контекстно-свободными грамматиками. Показано, что полученный алгоритм повышает производительность поиска структур, заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Описанный алгоритм и генератор синтаксических анализаторов реализованы в расках проекта YaccConstructor на языке программирования F#. Исходный код доступен в репозитории проекта: `https://github.com/YaccConstructor/YaccConstructor`.

Одним из методов для описания семантики языка являются атрибутные грамматики, но они не поддержаны в описанном алгоритме. Опубликовано несколько работ о подклассе атрибутных ECFG (например [4]), тем не менее нет общего решения для произвольных ECFG. Таким образом, поддержка атрибутных расширенных контекстно-свободных грамматик и подсчёт семантики может быть дальнейшим развитием данной работы.

Список литературы

1. ANTLR Project website. — `http://www.antlr.org/`.
2. Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // International Conference on Compiler Construction / Springer. — 2015. — P. 89–108.
3. Aho Alfred V, Hopcroft John E. The design and analysis of computer algorithms. — Pearson Education India, 1974.
4. Alblas Henk, Schaap-Kruseman Joos. An attributed ELL (1)-parser generator // International Workshop on Compiler Construction / Springer. — 1990. — P. 208–209.
5. Bison Project website. — `https://www.gnu.org/software/bison/`.
6. Breveglieri Luca, Crespi Reghizzi Stefano, Morzenti Angelo. Shift-Reduce Parsers for Transition Networks // Language and Automata Theory and Applications: 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings / Ed. by Adrian-Horia Dediu, Carlos Martín-Vide, José-Luis Sierra-Rodríguez, Bianca Truthe. — Cham : Springer International Publishing, 2014. — P. 222–235. — ISBN: 978-3-319-04921-2. — Access mode: `http://dx.doi.org/10.1007/978-3-319-04921-2_18`.
7. Breveglieri Luca, Reghizzi Stefano Crespi, Morzenti Angelo. Shift-reduce parsers for transition networks // International Conference on Language and Automata Theory and Applications / Springer. — 2014. — P. 222–235.
8. Brüggemann-Klein Anne, Wood Derick. On predictive parsing and extended context-free grammars // International Conference on Implementation and Application of Automata / Springer. — 2002. — P. 239–247.

9. Bruggemann-Klein Anne, Wood Derick. The parsing of extended context-free grammars. — 2002.
10. Heckmann Reinhold. An efficient ELL (1)-parser generator // Acta Informatica. — 1986. — Vol. 23, no. 2. — P. 127–148.
11. Heilbrunner Stephan. On the definition of ELR (k) and ELL (k) grammars // Acta Informatica. — 1979. — Vol. 11, no. 2. — P. 169–176.
12. Hemerik Kees. Towards a Taxonomy for ECFG and RRPG Parsing // International Conference on Language and Automata Theory and Applications / Springer. — 2009. — P. 410–421.
13. An n log n algorithm for minimizing states in a finite automaton : Rep. / DTIC Document ; Executor: John Hopcroft : 1971.
14. Lee Gyung-Ok, Kim Do-Hyung. Characterization of extended LR (k) grammars // Information processing letters. — 1997. — Vol. 64, no. 2. — P. 75–82.
15. Morimoto Shin-ichi, Sassa Masataka. Yet another generation of LALR parsers for regular right part grammars // Acta informatica. — 2001. — Vol. 37, no. 9. — P. 671–697.
16. Nawrocki Eric P, Eddy Sean R. Infernal 1.1: 100-fold faster RNA homology searches // Bioinformatics. — 2013. — Vol. 29, no. 22. — P. 2933–2935.
17. Purdom Jr Paul Walton, Brown Cynthia A. Parsing extended LR (k) grammars // Acta Informatica. — 1981. — Vol. 15, no. 2. — P. 115–127.
18. Quantifying variances in comparative RNA secondary structure prediction / James WJ Anderson, Ádám Novák, Zsuzsanna Sükösd et al. // BMC Bioinformatics. — 2013. — Vol. 14, no. 1. — P. 149.
19. Ragozina Anastasiya. GLL-based relaxed parsing of dynamically generated code : Master's Thesis / Anastasiya Ragozina ; SpBU. — 2016.
20. Reconstructing 16S rRNA genes in metagenomic data / Cheng Yuan, Jikai Lei, James Cole, Yanni Sun // Bioinformatics. — 2015. — Vol. 31, no. 12. — P. i35–i43.
21. Rekers Joan Gerard. Parser generation for interactive environments : Ph. D. thesis / Joan Gerard Rekers ; Universiteit van Amsterdam. — 1992.
22. Scott Elizabeth, Johnstone Adrian. GLL parsing // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189.

23. Scott Elizabeth, Johnstone Adrian. GLL parse-tree generation // Science of Computer Programming. — 2013. — Vol. 78, no. 10. — P. 1828–1844.
24. Scott Elizabeth, Johnstone Adrian. Structuring the GLL parsing algorithm for performance // Science of Computer Programming. — 2016. — Vol. 125. — P. 1–22.
25. Scott Elizabeth, Johnstone Adrian, Economopoulos Rob. BRNGLR: a cubic Tomita-style GLR parsing algorithm // Acta informatica. — 2007. — Vol. 44, no. 6. — P. 427–461.
26. Tellier Isabelle. Learning recursive automata from positive examples // Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle. — 2006. — Vol. 20, no. 6. — P. 775–804.
27. Thompson Ken. Programming Techniques: Regular Expression Search Algorithm // Commun. ACM. — 1968. — Jun. — Vol. 11, no. 6. — P. 419–422. — Access mode: `http://doi.acm.org/10.1145/363347.363387`.
28. Wheeler Travis J, Eddy Sean R. nhmmer: DNA homology search with profile HMMs // Bioinformatics. — 2013. — P. btt403.
29. Wirth Niklaus. Extended Backus-Naur Form (EBNF) // ISO/IEC. — 1996. — Vol. 14977. — P. 2996.
30. Xander: employing a novel method for efficient gene-targeted metagenomic assembly / Qiong Wang, Jordan A. Fish, Mariah Gilman et al. // Microbiome. — 2015. — Vol. 3, no. 1. — P. 32. — Access mode: `http://dx.doi.org/10.1186/s40168-015-0093-6`.
31. YaccConstructor [Электронный ресурс]. — Режим доступа: `https://github.com/YaccConstructor/YaccConstructor` (дата обращения: 11.05.2015).

Компиляция сертифицированных F*-программ в робастные Web-приложения

Полубелова Марина Игоревна

Санкт-Петербургский государственный университет
polubelovam@gmail.com

Аннотация Одним из способов повышения надежности систем является использование сертифицированного программирования для создания и верификации программного обеспечения. В рамках данного подхода реализация выполняется на языке с богатой типовой системой, статически гарантирующей некоторые свойства программы. Такая программа впоследствие транслируется в исполняемый на целевом устройстве язык программирования. Такой подход используется, например, для верификации криптографических примитивов в проекте HACL*, выполненном на языке программирования F*. В данной работе целевым языком для трансляции F*-программ выбран язык JavaScript, поддерживаемый всеми современными веб-браузерами. В работе предложены правила трансляции из F* в JavaScript, сохраняющие аннотации типов. Благодаря этой особенности возможно использовать инструмент Flow для дополнительной проверки полученной в результате трансляции программы.

Введение

В последнее время происходит стремительный рост количества устройств, подключаемых к Интернет. Взаимодей-

ствие между этими устройствами осуществляется с использованием различных протоколов. В зависимости от целей и потребностей пользователей Интернет-устройств формулируются различные требования к таким протоколам. Одно из важных требований является обеспечение конфиденциальности и целостности пользовательских данных. Например, при оплате онлайн-покупок банковской картой покупатель хочет быть уверен в том, что данные его карты не попадут к злоумышленникам, а банк получит именно те данные, которые были введены пользователем. Безопасность соединения таких Интернет-устройств зависит от используемых криптографических протоколов и кода Веб-приложений, который чаще всего создается на JavaScript [5]. Поэтому необходимо гарантировать безопасность используемых протоколов и Веб-приложений, а для последних еще и робастность (robustness). Веб-приложение является робастным, если оно продолжает работу даже при неверных входных данных, а не завершается аварийно.

Для работы Веб-приложений широко используется криптографический протокол Transport Layer Security (TLS) [3], который обеспечивает защищенную передачу данных. Для установки соединения между клиентом и сервером протокол сначала выполняет процедуру подтверждения сеанса связи, которая включает в себя согласование используемых алгоритмов шифрования и хэш-функций, валидацию сертификата сервера. После этого протокол устанавливает безопасное соединение, которое обеспечивает конфиденциальность передаваемых данных. Для сохранения целостности и аутентификации сообщений используются так называемые коды аутентификации — дополнительная информация, получаемая на основе передаваемых данных и позволяющая доказать, что сообщение не изменилось и не было подделано. Несмотря на продолжительное время разработки криптографических библиотек (например, библиотека OpenSSL [19] разрабатывается с 1998 года), они остаются подвержены хакерским атакам. При этом многие атаки используют ошибки, допущенные в программном обес-

печении. Например, ставшая известной в 2014 году уязвимость Heartbleed [36] в криптографической библиотеке OpenSSL позволяла несанкционированное чтение памяти на сервере или на клиенте.

Итак, существует необходимость в создании надежных систем, которые будут устойчивы к хакерским атакам. С одной стороны, для этого нужны новые протоколы, с другой — необходимы новые подходы к разработке программного обеспечения. Одним из таких подходов является использование сертифицированного программирования [1] для создания и верификации программ, которые затем транслируются в целевой язык. Сертифицированное программирование позволяет доказывать, что программа соответствует своему формальному описанию. Данный процесс происходит статически, что позволяет гарантировать, что программа всегда работает так, как указано в ее спецификации. В качестве инструментов для создания сертифицированных программ используются такие инструменты как Coq [22], Agda [20], F* [4], Idris [27] и другие. При этом необходимо гарантировать, что весь стек от создания программы до получения целевого кода является корректным и безопасным, так же, как и полученная в результате этого программа.

Данный подход применяется в проекте Everest [24], который нацелен на создание высокопроизводительной, соответствующей спецификациям, реализации протокола HTTPS. Данные в этом протоколе передаются поверх криптографического протокола TLS. Данный проект включает в себя подпроекты miTLS [32] и HACL* [16], которые посвящены верификации, соответственно, протокола TLS и криптографических примитивов. Для создания и верификации программ выбран функциональный язык программирования F* [25]. С одной стороны, этот язык обладает богатой системой типов, которая включает в себя зависимые (dependent types) и уточняющие типы (refinement types) [1]. С другой стороны, F* позволяет доказывать многие свойства программы, например, что

эффективная реализация программы соответствует своей спецификации. Для F* разработаны механизмы извлечения верифицированного кода в программы на OCaml и C [17]. Данный подход обеспечивает относительную независимость процесса создания программы от целевой платформы, позволяя использовать проверифицированный код во многих областях.

В данной работе целевым языком для компиляции F*-программ выбран JavaScript. Данный язык поддерживается всеми современными Веб-браузерами, не требуя установки дополнительного программного обеспечения. В данной работе предложены правила трансляции с языка F* в JavaScript (ECMAScript 6 [33]), гарантирующие сохранение аннотации типов. Это позволяет эффективно использовать инструмент Flow [26] (статический анализатор типов для JavaScript) для дополнительной проверки оттранслированной программы.

Данная работа была инициирована в рамках стажировки автора в INRIA Paris под руководством Karthikeyan Bhargavan. Проводимые исследования были поддержаны грантом JetBrains Research.

1 Постановка задачи

Целью данной работы является разработка инструмента для компиляции сертифицированных F*-программ в робастные Web-приложения на JavaScript. Для ее достижения были поставлены следующие задачи:

- сформулировать правила трансляции с языка F* на JavaScript, гарантирующие сохранение аннотаций типов;
- выполнить реализацию предложенного подхода;
- провести экспериментальное исследование реализованного инструмента.

2 Обзор

В данном обзоре описаны исходный и целевой языки трансляции, а именно, языки программирования F* и JavaScript. Описан инструмент Flow, используемый в работе, а также приведено обоснование выбора данного инструмента. Приведено также описание существующих инструментов для трансляции OCaml-программ в JavaScript-приложения. В конце обзора дано описание проекта HACL*, который является контекстом данной работы и из которого были взяты тестовые данные для проведения экспериментального исследования инструмента, реализованного в рамках данной работы.

2.1 Язык F*

В данной работе исходным языком для трансляции программ является функциональный язык программирования F* [4, 15, 25], который ориентирован на верификацию программ. Система типов F* включает в себя полиморфизм, уточняющие и зависимые типы (refinement and dependent types), monadic effects и вычисление слабейших предусловий. Данный язык позволяет создавать и верифицировать программы, то есть доказывать, что программа соответствует своей спецификации. Для этого данный инструмент использует SMT-решатель Z3 [30] и доказательства, написанные пользователем. Проверифицированный код можно извлечь в программы на OCaml и C [17].

Для того чтобы продемонстрировать основные особенности языка F*, рассмотрим пример вычисления факториала (см. листинг 1).

Для функции `fact` система типов языка F* выведет тип $int \rightarrow int$. Данная программа работает корректно для всех неотрицательных чисел, но если вызвать данную функцию от отрицательного аргумента, например, `fact -2`, то программа зациклится. Чтобы избежать такой ситуации, можно вве-

Listing 1 Функция вычисления факториала

```
let rec fact n =
    if n = 0 then 1 else n * (fact (n - 1))
```

сти ограничения на входные параметры функции. Сделать это можно с использованием уточняющих типов (см. листинг 2).

Listing 2 Функция вычисления факториала с эффектом *Tot*

```
val fact: x:int{x >= 0} -> Tot int
let rec fact n =
    if n = 0 then 1 else n * (fact (n - 1))
```

Уточняющие типы имеют вид $x : t\{phi(x)\}$, где элемент x принадлежит к типу t и удовлетворяет предикату $phi(x)$. В данном примере таким типом является `x:int{x >= 0}`. В этом же примере появилось ключевое слово *Tot* (total), которое отражает эффект вычисления функции. Данный эффект (*Tot t*) гарантирует, что функция всегда завершает работу, а полученный результат имеет тип t. Когда в программе происходит вызов данной функции, то при верификации проверяется, что аргументы вызывающей функции удовлетворяют ее условиям.

Для доказательства корректности программ можно использовать леммы, которые не имеют вычислительного эффекта и всегда возвращают `unit`, но они позволяют доказывать многие свойства программы. При этом формулировка леммы “находится” в типах, а само доказательство приводится в теле функции. Например, можно доказать, что результат вычисления функции факториала числа всегда является положительным числом (см. листинг 3).

Доказывается данное свойство индукцией по числу x. В данном примере используется ключевое слово *GTot* (ghost total), которое как раз и означает, что функция не имеет вы-

Listing 3 Результат вычисления факториала является положительным числом

```
val fact_property: x:int{x>=0} -> GTot (u:unit{fact x > 0})
let rec fact_property x =
   match x with
   | 0 -> ()
   | _ -> fact_property (x - 1)
```

числительного эффекта и всегда возвращает тип `unit` (ghost-вычисления). При трансляции таких функций сохраняется только их сигнатура (уточняющие и зависимые типы заменяются на близкие им типы в целевом языке), при этом тело самой функции удаляется. Результат трансляции данного примера представлен в листинге 4.

Listing 4 Результат трансляции функции *fact_property*

```
let fact_property (x:int) = ()
```

В данном примере используется зависимый тип `u:unit{fact x > 0}`, который в общем случае имеет вид $x : t_1 \to \ldots \to x_n : t_n[x_1 \ldots x_{n-1}] \to E\ t[x_1 \ldots x_n]$. Нотация $t[x_1 \ldots x_{n-1}]$ означает, что переменные $x_1 \ldots x_{n-1}$ могут свободно входить в аргумент t. E содержит эффект вычисления тела функции.

Данная функция может быть записана в виде, представленном в листинге 5, где ключевые слова `requires` и `ensures` соответствуют предусловию и постусловию леммы.

Как уже было сказано, если не накладывать ограничения на входные параметры функции, то при вызове функции с отрицательным аргументом программа зациклится. Данный эффект можно отобразить с использованием ключевого слова *Dv* (см. листинг 6).

Listing 5 Результат вычисления факториала является положительным числом

```
val fact_property: x:int -> Lemma
    (requires (x >= 0))
    (ensures (fact x > 0))
let rec fact_property x =
   match x with
   | 0 -> ()
   | _ -> fact_property (x - 1)
```

Listing 6 Функция вычисления факториала с эффектом Dv

```
val fact : int -> Dv int
let rec fact n =
    if n = 0 then 1 else n * (fact (n - 1))
```

Функцию вычисления факториала числа также можно написать с использованием ссылочных переменных (см. листинг 7), которые фактически являются указателями. Для обращения к хранимому значению используется нотация $!x$, для изменения хранимого значения — $x1 := x2$. Ключевое слово ST (stateful) означает, что выполнение функции происходит с использованием кучи, то есть могут выполняться операции чтения и записи, создания новых ссылочных переменных и т.д. Так как для вычислений используется куча, то необходима модель памяти, которая позволит доказывать необходимые свойства. Например, что два разных указателя не ссылаются на одну и ту же память (alias analysis). В данном примере $upd\ h0\ r1\ x$ означает, что нужно обновить значение переменной по адресу $r1$ в куче $h0$ значением x. Конструкция $sel\ h0\ r1$ означает, что нужно выбрать значение, находящееся по адресу $r1$ в куче $h0$. Постусловие `ensures` утверждает, что существует такое x, что в результате выполнения функции `fact_st` в куче $h1$ будет содержаться результат выполнения функции `fact x` (так как $sel\ h0\ r1 = x$) по адресу $r2$.

Listing 7 Функция вычисления факториала с эффектом ST

```
type nat = x:int{x>=0}

val fact_st : r1:ref nat -> r2:ref nat -> ST unit
    (requires (fun h0 -> True))
    (ensures (fun h0 _ h1 ->
    exists x. h1 == (upd (upd h0 r1 x) r2 (fact (sel h0 r1)))))
let rec fact_st r1 r2 =
  let x1 = !r1 in
  if x1 = 0
  then r2 := 1
  else
     r1 := x1 - 1;
     fact_st r1 r2;
     let x2 = !r2 in
     r2 := x2 * x1
```

Самым общим эффектом вычисления является эффект ML, который включает в себя все остальные эффекты. Пример использования такого эффекта представлен в листинге 8.

Listing 8 Функция вычисления факториала с эффектом ML

```
val fact : int -> ML int
let rec fact n =
    if n = 0 then 1 else n * (fact (n - 1))
```

Таким образом, эффекты $\{Tot, ML, Dv, ST\}$ образуют решетку, в которой $Tot < Dv < ML$ и $Tot < ST < ML$. Пользователи также могут добавлять свои эффекты вычисления.

2.2 Особенности языка JavaScript

JavaScript является популярным языком программирования, используемым для разработки Веб-приложений [14]. Данный язык является динамически типизированным и прототип-ориентированным и не имеет стандартной нотации хранения переменных [5]. Все это привносит ряд сложностей при работе: семантика языка сложна для понимания, а отсутствие типизации данных увеличивает количество возможных ошибок и затрудняет читаемость кода.

Осложняет процесс понимания работы языка JavaScript, например, принцип хранения переменных. Чтобы показать его особенности, приведем пример из [5]:

Listing 9 Пример функции, демонстрирующий особенности хранения переменных

```
x = null; y = null; z = null;
f = function(w){x = v; v = 4; var v; y = v;};
v = 5; f(null); z = v;
```

Необходимо ответить на вопрос, какие значения будут иметь переменные `x`, `y` и `z` после завершения программы. Правильным ответом являются, соответственно, `undefined`, 4 и 5. Так как при выполнении программы внутри тела функции `f` все локальные переменные выносятся в начало блока и инициализируются значениями `undefined`, то функция `f` неявно имеет вид, представленный в листинге 10.

Перед вызовом функции `f` значение переменной `v` равно 5, однако внутри функции `f` также есть локальная переменная `v`. Так как все локальные переменные выносятся в начало блока и инициализируются значениями `undefined`, то переменная `x` имеет значение `undefined`. При этом выполнение функции `f` повлияло на значения глобальных переменных `x` и `y`.

Listing 10 Неявный вид функции, представленный в листинге 9

```
f = function(w){
    var v = undefined;
    x = v; v = 4; y = v;
}
```

Одним из подходов к созданию более надежных JavaScript-программ является написание программ на языке со строгой типизацией с последующей их трансляции в JavaScript [34]. Такой подход обладает рядом преимуществ. Во-первых, исходный язык для написания программ является строго типизированным, что позволяет обнаруживать ошибки на этапе компиляции программы. Во-вторых, данный подход позволяет избежать ряд ручных проверок на соответствие типов. В-третьих, благодаря более высокому уровню абстракции программы, повышается уровень ее понимания и отсутствует необходимость в запоминании типов используемых переменных.

Другим подходом является создание нового языка, который близок к JavaScript и для которого уже можно разработать инструменты статического анализа кода. Например, язык программирования TypeScript [37] и инструмент Flow [26], который является статическим анализатором для JavaScript.

В данной работе при трансляции F*-программ в JavaScript-приложения используется последняя версия спецификации языка JavaScript (ECMAScript 6), в которой представлены новые конструкции языка. Например, использование конструкций `let` и `const` для объявления переменных вместо конструкции `var` позволяет контролировать область видимости переменных. Использование стрелочных функций (arrow function) `(x) => {e}` позволяет корректно реализовать частичное применение функции к ее аргументам.

2.3 Инструмент Flow

Инструмент Flow [26] является статическим анализатором типов для JavaScript. Он позиционируется как средство для создания робастного и безопасного, с точки зрения системы типов, JavaScript-кода. Добавление типов к динамически типизированному языку имеет ряд преимуществ. Во-первых, это позволяет статически находить ошибки при проверке согласованности типов, например, `2 + "hi"`. Во-вторых, это помогает при документировании системы. В-третьих, позволяет в IDE включать функциональность, которая помогает при работе с кодом. Хотя добавление дополнительных проверок отражается на время разработки программы, разработчики инструмента Flow ввели несколько режимов работы с инструментом: не проверять код совсем, проверять код без типовой аннотации и проверять код с внесенным в него типовой информацией. Быстро проверять большую кодовую базу помогает следующий подход: параллельно обрабатывать каждый модуль программы, после чего соединять результаты в один, а также проверять только те файлы, которые были изменены. Возможность постепенного введения информации о типах в программу делает инструмент Flow применимым к уже существующим проектам.

Целью создания инструмента Flow является также обеспечение более точного информирования об ошибках, которые нарушают согласованность типов. При этом анализ согласованности типов обладает больше свойством soundness, чем completness. То есть инструмент информирует не только об ошибках, которые точно произойдут, но и о возможных ошибках, которые могут ими и не быть.

Сравнение Flow и TypeScript

Ниже приведены основные критерии, которые повлияли на выбор инструмента Flow. Данные о сравнении инструмента Flow с языком TypeScript взяты из источника [35].

– TypeScript — язык программирования, Flow — инструмент для статической проверки типов в JavaScript-программах,

в котором аннотация типов легко убирается из программы, используя плагин из babel.
- Flow имеет более богатую и точную (soundness) систему типов, чем TypeScript.
- Логика работы системы типов инструмента Flow близка к той, что используется в функциональных языках программирования.

Использование инструмента Flow в данной работе позволяет упростить правила трансляции типов, так как логика работы системы типов инструмента Flow близка к той, что используется в OCaml. При этом, если разработчик уверен, что программа правильно типизирована, то можно, не запуская инструмент Flow, удалить аннотацию типов из программы и получить готовое приложение, что сократит время разработки.

2.4 Инструменты трансляции программ на OCaml в JavaScript-приложения

В рамках данной работы были выдвинуты специальные требования к инструменту для компиляции F*-программ в JavaScript-приложения. Во-первых, необходимо гарантировать, что весь стек от создания программы до получения целевого кода является корректным и безопасным, как и полученная в результате этого программа. Во-вторых, F* позволяет при создании программы использовать конструкцию *assume*, которая описывает сигнатуру функцию без ее реализации. Например, конструкция *assume* может использоваться, если нет необходимости верифицировать функцию или реализация функции зависит от целевой платформы. При трансляции кода, помеченного конструкцией *assume*, порождаются функции без тела, поэтому необходимо предоставить конечному пользователю возможность подменять реализацию таких функций.

Для трансляции F*-программ в JavaScript-приложения можно использовать реализованный в рамках проекта F* механизм извлечения верифицированного кода в программы на OCaml, то есть трансляция F*-программ в JavaScript-приложения сведется к трансляции OCaml-программ в JavaScript-приложения. Существуют два инструмента для трансляции OCaml-программ в JavaScript-приложения. Первый инструмент js_of_ocaml [31] проводит трансляцию программ на уровне байт-кода, в результате чего получается эффективный код, который не сохраняет связь с исходным. Второй инструмент BuckleScript [21] проводит трансляцию программ на уровне лямбда-выражений и создает при этом читаемый, эффективный и модульный код.

Требованию безопасности инструментального стека существующие решения не удовлетворяют, так как ни один из инструментов не гарантирует, что полученная в результате трансляции программа является корректной. При этом многие оптимизации, сделанные в данных инструментах, нарушают согласованность типов. Например, трансляция булевого значения в числовое представление позволяет использовать функции, возвращающие булево значение, в арифметических выражениях. То есть, когда $true$ и $false$ транслируются, соответственно, в 1 и 0, то возможно использование, например, функции $exists\ l\ x$, которая проверяет вхождение элемента x в список l, в арифметическом выражении $(exists\ l\ x) + 5$, что семантически является некорректным, с точки зрения семантики целевого языка.

Для второго требования, которое заключается в предоставлении пользователям возможности подменять код некоторых функций на целевом языке, можно предложить два подхода. Первый подход позволяет предоставлять реализацию необходимых функций на языке OCaml, а второй подход — на языке JavaScript. Так как OCaml является промежуточным языком при трансляции F*-программ в JavaScript-приложения, то взаимодействие с еще одним языком программирования может

осложнить процесс получения целевого кода. При этом существующий подход в проекте FStarLang для получения OCaml-программ из F*-программ не позволяет использовать существующие инструменты для компиляции OCaml-программ в JavaScript. Это связано с тем, что та реализация, которая дается для некоторых стандартных F*-библиотек на OCaml использует вставки кода на C. Для того чтобы использовать второй подход, необходимо, чтобы полученный в результате трансляции код, сохранял связь с исходным кодом. Для этого подходит только инструмент BuckleScript, однако данный инструмент использует внутреннее представление для некоторых структур данных, что нужно учитывать при разработке программ. Например, внутреннее представление используется для списков и записей, так если в исходном коде список имел вид $1 :: 2 :: [3]$, то в целевом коде это будут вложенные списки $[1, [2, [3, 0]]]$.

Таким образом, существует необходимость в разработке инструмента для трансляции программ на F*, а именно подмножества языка OCaml, в JavaScript, который будет удовлетворять заявленным требованиям.

2.5 Проект HACL*

Проект HACL* (High-Assurance Cryptographic Library) [16] посвящен верификации криптографических примитивов, которые используются в качестве компонентов при построении криптографических протоколов. Для создания и верификации программ используется язык F*. Проверифицированный код извлекается в программы на OCaml и C. На текущий момент проверифицированы следующие примитивы:

- Stream ciphers: Chacha20, Salsa20, XSalsa20;
- MACs: Poly1305;
- Elliptic Curves: Curve25519;
- NaCl API: secret_box, box.

Целью проекта HACL* является создание эталонных реализаций (reference implementation) популярных криптографических примитивов и верификации их на безопасный доступ к памяти (memory safety), устойчивости к атакам по сторонним каналам (side-channel attack) и функциональной корректности. Данный проект является частью проекта Everest [24], который нацелен на создание проверифицированной версии протокола HTTPS. В этот проект также входят проекты, перечисленные ниже.

- Язык программирования F* [4], ориентированный на верификацию программ.
- Проект miTLS [32], который посвящен верификации протокола TLS. Для создания и верификации программ использовался язык F*.
- Проект HACL* [16], который посвящен верификации криптографических примитивов. Для создания и верификации программ использовался язык F*.
- KreMLin [17] — компилятор из подмножества языка F* в C.
- Язык программирования Vale [28] и инструмент Dafny [23] для создания и верификации криптографических примитивов на ассемблере.

Проект HACL* является контекстом данной работы, так как одним из дальнейших направлений работы является поддержка HACL* в качестве криптографической библиотеки для JavaScript. Из данного проекта были взяты тестовые данные для проведения экспериментального исследования инструмента, реализованного в рамках данной работы.

3 Правила трансляции с языка F* на JavaScript

В данном разделе описаны подход, который применяется к трансляции языка F* в JavaScript, и предложенные правила трансляции.

3.1 Описание подхода

Классическим подходом к трансляции программ является использование синтаксически управляемой трансляции [10]. Данный подход сводится к построению абстрактного синтаксического дерева (Abstract Syntax Tree, AST) исходной программы, которое затем необходимо обойти, чтобы применить правила трансляции к каждой вершине построенного дерева. Результатом данного обхода является AST целевой программы, по которому уже можно получить код целевой программы.

Данный подход применяется в этой работе. Для того чтобы была возможность выполнять F*-программы, в проекте FStarLang [15] разработан механизм извлечения проверифицированного кода в OCaml. Данный механизм можно использовать для создания новых бэкендов для инструмента F*, как это уже было сделано для языка KreMLin, который является промежуточным языком для трансляции подмножества языка F* в C. Данный механизм используется и в данной работе, то есть трансляция языка F* в JavaScript сводится к трансляции подмножества языка OCaml в JavaScript. При этом правила трансляции сохраняют аннотацию типов с целью эффективного использования инструмента Flow для статической проверки извлеченного из F* кода.

Трансляция языка F* в JavaScript происходит с использованием ML AST программы, полученной в результате трансляции F* в OCaml. Процесс получения результирующего Flow AST [9] сводится к обходу ML AST с целью применения предложенных правил трансляции к каждой вершине этого дерева. Затем происходит кодогенерация, то есть процесс получения программы, которая может быть проверена инструментом Flow. Ниже представлена модель трансляции F*-программы в JavaScript-приложение. Последнее преобразование удаляет аннотацию типов и заменяет ES-стиль на CommandJS-стиль.

$$\text{.fst} \rightarrow \text{F* AST} \rightarrow \text{ML AST} \rightarrow \text{Flow AST} \rightarrow \text{.flow} \rightarrow \text{.js}$$

Такой подход обладает рядом преимуществ. Во-первых, AST содержит больше информации об исходном коде, которую можно использовать эффективно при трансляции. Например, существующие инструменты для компиляции программ на OCaml в JavaScript-приложения, а именно, инструменты js_of_ocaml [31] и BuckleScript [21] проводят трансляцию, соответственно, с байт-кода OCaml и лямбда-выражений OCaml. При таком подходе удаляется информация о типах и становится сложнее сохранить связь с исходным кодом. Во-вторых, такой подход позволяет доказать корректность трансляции, используя классические подходы.

3.2 Правила трансляции

В данном разделе описаны предложенные правила трансляции сертифицированных F*-программ в робастные Веб-приложения. В предлагаемых правилах трансляции можно выделить следующие четыре группы.

1. Правила трансляции констант.
2. Правила трансляции выражений.
3. Правила трансляции выражений для сопоставления с образцом.
4. Правила трансляции типов.

Для трансляции программ используется окружение ρ (environment) — это упорядоченный список, элементы которого имеют структуру, представленную в листинге 11, где $names$ — список используемых переменных, $module\_names$ — имя текущего модуля, $import\_module\_names$ — список имен модулей, которые используются в текущем модуле.

Так как модель хранения переменных в JavaScript похожа на принцип работы стека (stack frames), то при трансляции языка OCaml в JavaScript окружение ρ будет его моделировать. То есть каждый новый блок (scope) в программе соответствует новому элементу стека (frame). После выхода из блока,

Listing 11 Окружение, которое используется правилами трансляции

```
type env_t = {
  names: list<name>;
  module_name: string;
  import_module_names: list<string>;
}
```

снимается элемент со стека. Новый элемент добавляется в список *names*, когда встречается операция создания переменной в программе. Для того чтобы уменьшить количество блоков в целевой программе, проверяется вхождение новой переменной в *names* и если оно в нем содержится, то новый блок создается, иначе нет. В список *import_module_names* добавляется новый элемент, когда встречается конструкция типа *name*, которая содержит в себе информацию, в каком модуле она определена. Если название модуля не совпадает с текущим и не содержится в списке *import_module_names*, то имя этого модуля добавляется в этот список. При этом с данным списком не происходят операции удаления на протяжении трансляции всего модуля. Этот список будет включен в начало файла при печати Flow AST как список модулей, которые нужно импортировать в данный модуль.

Ниже представлено описание каждой группы правил трансляции.

1. Правила трансляции констант В JavaScript нет различий между типами *integer* и *float*, все числовые переменные является объектами типа *number*, который позволяет работать с числами. Во всех остальных случаях существуют аналоги OCaml-констант в JavaScript, которые не требуют особых преобразований. Правила трансляции типов констант представлены в листинге 3.19. Трансляция констант не изменяет содержимое окружения ρ.

2. Правила трансляции выражений Правила трансляции выражений делятся на две части. Первая часть отвечает за трансляцию OCaml-выражений в JavaScript-выражения, а вторая часть — за трансляцию OCaml-выражений в JavaScript-операторы. Связано это с тем, что грамматические конструкции языка OCaml включают в себя только выражения (expressions) (см. таблицы 3.15 и 3.16), в то время как синтаксис языка JavaScript содержит в себя как выражения, так и операторы (statements) (см. таблицы 3.17 и 3.18). В листинге 12 приведен пример OCaml-программы, в которой условное выражение `if-then-else` используется внутри арифметического выражения.

Listing 12 Конструкция $if - then - else$ внутри арифметического выражения

```
let f x b = x + (if b then 5 else 0)
```

Так как синтаксис языка JavaScript не позволяет использовать операторы внутри выражений, то при трансляции необходимо учитывать ситуации, когда транслируемое OCaml-выражение не являются выражением в целевом языке. Одним из подходов к решению данной проблемы является создание новой переменной (fresh variables), которая будет содержать в себе результат выполнения таких выражений. Другим подходом предлагается использование анонимных функций, которые в новой спецификации языка JavaScript (ECMAScript 6) являются выражением. В данной работе реализован первый подход, то есть с использованием новых переменных, так как второй подход порождает более громоздкий код. Поэтому результатом трансляции рассмотренного выше примера является программа, код которой представлен в листинге 13.

Таким образом, правила трансляции для выражений используют два типа функций трансляции. Первый тип $[\ ]_{\rho}$

Listing 13 Результат трансляции кода, представленного в листинге 12

```
let f = (x) => ((b) => {
    let res;
    let fv;
    if (b) {fv = 5;} else {fv = 0;}
    res = x + fv;
    return res;
})
```

используется, когда происходит трансляция $expr\_ml \rightarrow expr\_js$ и второй тип $[\![\ ]\!]_\rho\ \_\_$, когда происходит трансляция $expr\_ml \rightarrow stmt\_js$. Правила трансляции OCaml-выражений представлены в листингах 3.20 и 3.21. Обозначение le используется для списка вида $e_1, e_2, \ldots, e_n$. Правило трансляции такого списка имеет следующий вид: $[\![\ (e_1,\ \ldots\ , e_n)\ ]\!]_\rho = ([\![\ e_1\ ]\!]_\rho, \ldots, [\![\ e_n\ ]\!]_\rho)$.

Правила трансляции, указанные в листинге 3.21, имеют следующую интерпретацию: транслируем выражением e_{ml}, после чего результат трансляции записываем в переменную x_{js} и дальше выполняем s_{js} (либо ничего не делаем, либо продолжаем трансляцию).

Использование конструкций *let* и *const* при создании новых переменных в целевом языке порождает ряд дополнительных проверок при осуществлении трансляции, чтобы обеспечить правильную область видимости переменных. Например, необходимо проверять, встречается ли в параметрах вызова функции переменная, имя которой совпадает с именем переменной, которой присваивается результат, возвращаемый данной функцией:

Для того чтобы данная программа была корректна, с точки зрения JavaScript, необходимо ввести новую переменную, в которой будет сохранено значение переменной `l`, при этом сделать это надо до начала блока:

```
let g l =
    let l = f l in
    l + 3
```

```
let g = (l) => {
    let res;
    let l1 = l; {
        let l = f(l1);
        res = l + 3;
    }
    return res;
}
```

Резюмируя все вышесказанное, правило трансляции конструкции создания новой локальной переменной x имеет следующий вид:

if cond then s_js
if $x \in \rho$ then { let x = ... }
if let x = f(x) then let fv_x = x; { let x = f(fv_x); ... }
otherwise x

На этапе трансляции некоторых конструкций можно осуществить их замену на конструкции или функции из целевого языка для более эффективной работы оттранслированной программы. Например, в правиле $[\![C\ e_1\ \ldots\ e_n]\!]_\rho$ в зависимости от значения имени конструктора C используется замена, указанная в листинге 3.14.

OCaml	*Flow*
Option t	$= t?$
List, Array, Seq	$= Array$
Tuple	$= Array$
Record	$= Object$

Листинг 3.14: Частные случаи трансляции конструктора

3. Правила трансляции выражений для сопоставления с образцом Правила трансляции выражений для сопоставления с образцом представлены в листинге 3.23 и имеют следующую интерпретацию: транслируем выражение p_{ml}, сравниваем полученный результат с выражением e_{js}, если они совпадают, то выполняем дальше s_1_then, иначе s_2_else.

В правилах, указанных в листинге 3.23, можно избавиться от повторения конструкции s_2. Пример одного из подхода, позволяющий это сделать, приведен в листинге 3.24. Аналогично можно поступить и с конструкциями $C\ e_1\ \ldots\ e_n$, $(g_1 : e_1,\ \ldots\ g_n : e_n)$, $(e_1,\ \ldots,\ e_n)$.

4. Правила трансляции типов Синтаксис типов исходного языка представлен в листинге 3.25 , целевого языка — в листинге 3.26. Правила трансляции типов представлены в листинге 3.27. В зависимости от значения имени конструктора можно ввести замену, согласованную с листингом 3.14.

$e \in$ **Exp** выражения (expressions)
$x, f \in$ **Var** переменные
$c \in$ **Const** константы
$p \in$ **Pattern** выражения для сопоставления с образцом (pattern matching)

Листинг 3.15: Используемые обозначения

$c ::=$	n	числа (int, float)
	b	булевы константы (true, false)
	string	строки
	()	значение типа unit
$e ::=$	c	константы
	x	переменные
	name	value-name
	let $x = e_1$ in e_2	локальное связывание
	$f\ x$	вызов функции
	fun x => e	определение функции
	match e with $\mid p_i -> e_i$	сопоставление с образцом
	$C\ e_1\ \ldots\ e_n$	конструктор
	$e_1; \ldots; e_n$	последовательность
	$(e_1, \ldots, e_n)$	кортеж
	$(g_1 : e_1, \ldots, g_n : e_n)$	запись
	$(e, name)$	проекция
	if e then e_1 else e_2	условное выражение
$p ::=$	_	символ-wildcard
	c	константы
	x	переменные
	$C\ p_1\ \ldots\ p_n$	конструктор
	$p_1 \mid p_2 \mid \ldots \mid p_n$	альтернативы шаблонов
	$(g_1 : p_1, \ldots, g_n : p_n)$	запись
	$(p_1, \ldots, p_n)$	кортеж
	p when e	“охранные” выражения

Листинг 3.16: Синтаксис используемого в данной работе подмножества языка OCaml

$e \in \mathbf{Exp}$ выражения (expressions)
$s \in \mathbf{Stmt}$ операторы (statements)
$x, f \in \mathbf{Var}$ переменные
$c \in \mathbf{Const}$ константы
$a \in \mathbf{lvalue}$ левая часть присваивания

Листинг 3.17: Используемые обозначения

$c ::=$	n	числа (number)
	b	булевы константы (true, false)
	string	строки
	null	null
$a ::=$	x	переменные
	$x_l.f$	обращение по имени поля к объекту (object)
	$x_l[e]$	обращение по индексу к массиву
$e ::=$	c	константы
	x	переменные
	$x.f$	обращение по имени поля к объекту
	$x[e]$	обращение по индексу к массиву
	$\{g_1 : e_1, \ldots, g_n : e_n\}$	объект (object)
	$[e_1, \ldots, e_n]$	массив
	$f(x)$	вызов функции
	$(x) \Rightarrow e$	стрелочная функция
	$a = e$	присваивание
	$e_1; \ldots; e_n$	последовательность
$s ::=$	if e then s_1 else s_2	условное выражение
	$s_1; \ldots; s_n$	последовательность
	let $x = e$	локальное определение
	return e	return-выражение

Листинг 3.18: Синтаксис используемого в данной работе подмножества языка JavaScript

$$
\begin{array}{ll}
[\ c_{ml}\ ] & = c_{js} \\
[\ unit\ ] & = null \\
[\ bool\ ] & = bool \\
[\ string\ ] & = string \\
[\ int\ ] & = number \\
[\ float\ ] & = number
\end{array}
$$

Листинг 3.19: Правила трансляции типов OCaml-констант в типы JavaScript-констант

$$
\begin{array}{ll}
[\ e_{ml}\ ]_\rho & = e_{js} \\
[\ c\ ]_\rho & = c \\
[\ name\ ]_\rho & = name\_js \\
[\ f\ x\ ]_\rho & = f_{js}([\ x\ ]_\rho) \\
[\ f\ x\ ]_\rho & = [\ f\ ]_\rho\ ([\ x\ ]_\rho) \\
[\ (e_1, \ldots, e_n)\ ]_\rho & = [[\ e_1\ ]_\rho, \ldots, [\ e_n\ ]_\rho]_\rho \\
[\ (g_1 : e_1, \ldots, g_n : e_n)\ ]_\rho & = \{\_tag : \text{“}Record\text{“}, \_g_1 :[\ e_1\ ]_\rho, \ldots, \_g_n :[\ e_n\ ]_\rho\} \\
[\ C\ e_1 \ldots e_n\ ]_\rho & = \{\_tag : \text{“}C\text{“}, \_1 :[\ p_1\ ]_\rho, \ldots, \_n :[\ p_n\ ]_\rho\} \\
[\ e.name\ ]_\rho & = [\ e\ ]_\rho\ .\_name
\end{array}
$$

Листинг 3.20: Правила трансляции OCaml-выражений в JavaScript-выражения

$$
\begin{array}{ll}
[\![\, e_{ml} \,]\!]_\rho \;\; x_{js} \;\; s_{js} & = s_{js} \\
[\![\, c \,]\!]_\rho \;\; x \;\; s & = \text{let } x \;=\; [\, c \,]_\rho;\; s \\
[\![\, x \,]\!]_\rho \;\; y \;\; s & = \text{let } y \;=\; [\, x \,]_\rho;\; s \\
[\![\, \text{let } x = e_1 \; in \; e_2 \,]\!]_\rho \;\; y \;\; s & = [\![\, e_1 \,]\!]_\rho \; [\, x \,]_\rho \;\; ([\![\, e_2 \,]\!]_{\rho_1} \;\; y \;\; s) \\
[\![\, f \;\; x \,]\!]_\rho \;\; y \;\; s & = \text{let } y \;=\; f_{js}([\, x \,]_\rho);\; s \\
[\![\, f \;\; x \,]\!]_\rho \;\; y \;\; s & = [\![\, x \,]\!]_\rho \;\; fv\_x \;\; ([\![\, f \;\; fv\_x \,]\!]_{\rho_1} \;\; y \;\; s) \\
[\![\, \text{fun } x \;=\!> \; e \,]\!]_\rho \;\; f \;\; s & = \text{let } f \;=\; ([\, x \,]_\rho) \\
 & \quad =\!> \{[\![\, e \,]\!]_{\rho_1} \;\; \_res \;\; (return \;\; \_res)\};\; s \\
[\![\, \text{match } e \text{ with } \mid p_i \; -\!> \; e_i \,]\!]_\rho \;\; x \;\; s & = [\![\, e \,]\!]_\rho \;\; fv\_e \\
 & \quad (translate\_match \;\; fv\_e \;\; lp \;\; le \;\; x \;\; \rho_1);\; s \\
[\![\, C \;\; e_1 \;\; \ldots \;\; e_n \,]\!]_\rho \;\; x \;\; s & = \text{let } x \;=\; [\, C \;\; e_1 \;\; \ldots \;\; e_n \,]_\rho;\; s \\
[\![\, C \;\; e_1 \;\; \ldots \;\; e_n \,]\!]_\rho \;\; x \;\; s & = [\![\, le \,]\!]_\rho \;\; fv\_le \;\; ([\![\, C \;\; fv\_le \,]\!]_{\rho_1} \;\; x \;\; s) \\
[\![\, e_1; \ldots; e_n \,]\!]_\rho \;\; x \;\; s & = [\![\, e_1 \,]\!]_\rho \;\; \_ \;\; ([\![\, e_2 \,]\!]_{\rho_1} \;\; \_ \\
 & \quad (\ldots ([\![\, e_n \,]\!]_{\rho_{n-1}} \;\; x \;\; s))) \\
[\![\, (e_1, \ldots, e_n) \,]\!]_\rho \;\; x \;\; s & = \text{let } x \;=\; [\, (e_1, \ldots, e_n) \,]_\rho;\; s \\
[\![\, (e_1, \ldots, e_n) \,]\!]_\rho \;\; x \;\; s & = [\![\, le \,]\!]_\rho \;\; fv\_le \;\; ([\![\, (fv\_le) \,]\!]_{\rho_1} \;\; x \;\; s) \\
[\![\, (g_1 : e_1, \ldots, g_n : e_n) \,]\!]_\rho \;\; x \;\; s & = \text{let } x \;=\; [\, (g_1 : e_1, \ldots, g_n : e_n) \,]_\rho;\; s \\
[\![\, (g_1 : e_1, \ldots, g_n : e_n) \,]\!]_\rho \;\; x \;\; s & = [\![\, le \,]\!]_\rho \;\; fv\_le \;\; ([\![\, (lg : fv\_le) \,]\!]_{\rho_1} \;\; x \;\; s) \\
[\![\, e.name \,]\!]_\rho \;\; x \;\; s & = \text{let } x \;=\; [\, e.name \,]_\rho;\; s \\
[\![\, \text{if } e \text{ then } e_1 \text{ else } e_2 \,]\!]_\rho \;\; x \;\; s & = [\![\, e \,]\!]_\rho \;\; \_cond \\
 & \quad (\text{if}(\_cond)\{[\![\, e_1 \,]\!]_\rho \;\; x \;\; None\} \\
 & \quad \text{else } \{[\![\, e_2 \,]\!]_\rho \;\; x \;\; None\});\; s
\end{array}
$$

Листинг 3.21: Правила трансляции OCaml-выражений в JavaScript-операторы

$$
\begin{array}{ll}
translate\_match \;\; fv\_e \;\; lp \;\; le \;\; x \;\; \rho & = [\![\, p_1 \,]\!]_\rho \;\; fv\_e \;\; ([\![\, e_1 \,]\!]_{\rho_1} \;\; x \;\; None) \\
 & \quad ([\![\, p_2 \,]\!]_\rho \;\; fv\_e \;\; ([\![\, e_2 \,]\!]_{\rho_2} \;\; x \;\; None) \;\; \ldots \\
 & \quad ([\![\, p_n \,]\!]_\rho \;\; fv\_e \;\; ([\![\, e_n \,]\!]_{\rho_n} \;\; x \;\; None) \\
 & \quad Exception))
\end{array}
$$

Листинг 3.22: Функция *translate_match*

$$
\begin{array}{ll}
[\![\, p_{ml} \,]\!]_\rho \;\; e_{js} \;\; s_1\_then \;\; s_2\_else & = s_{js} \\
[\![\, \_ \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = s_1 \\
[\![\, c \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = \text{if } (e \;==\; c) \; \{s_1\} \text{ then } \{s_2\} \\
[\![\, x \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = \text{let } x \;=\; e; \;\; s_1 \\
[\![\, C \;\; p_1 \;\; \ldots \;\; p_n \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = \text{if } (e.\_tag \;===\; \text{“}C\text{“}) \\
 & \quad [\![\, p_1 \,]\!]_\rho \;\; e.\_1 \; ([\![\, p_2 \,]\!]_\rho \;\; e.\_2 \\
 & \quad (\ldots([\![\, p_n \,]\!]_\rho \;\; e.\_n \;\; s_1 \;\; s_2)) \;\; s_2) \;\; s_2 \\
 & \quad \text{else } s_2 \\
[\![\, p_1 \mid p_2 \mid \ldots \mid p_n \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = [\![\, p_1 \,]\!]_\rho \;\; e \;\; s_1 \; ([\![\, p_2 \,]\!]_\rho \;\; e \;\; s_1 \\
 & \quad (\ldots([\![\, p_n \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2)) \\
[\![\, (g_1 : p_1, \ldots, g_n : p_n) \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = [\![\, p_1 \,]\!]_\rho \;\; e.\_g_1 \; ([\![\, p_2 \,]\!]_\rho \;\; e.\_g_2 \\
 & \quad (\ldots([\![\, p_n \,]\!]_\rho \;\; e.\_g_n \;\; s_1 \;\; s_2) \;\; s_2) \;\; s_2 \\
[\![\, (p_1, \ldots, p_n) \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = [\![\, p_1 \,]\!]_\rho \;\; e[1] \; ([\![\, p_2 \,]\!]_\rho \;\; e[2] \\
 & \quad (\ldots([\![\, p_n \,]\!]_\rho \;\; e[n] \;\; s_1 \;\; s_2) \;\; s_2) \;\; s_2 \\
[\![\, p \text{ when } g \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 & = [\![\, p \,]\!]_\rho \;\; e \\
 & \quad ([\![\, g \,]\!]_{\rho_1} \;\; \_x \; (\text{if } (\_x) \; \{s_1\} \text{ else } \{s_2\})) \;\; s_2
\end{array}
$$

Листинг 3.23: Правила трансляции сопоставления с образцом

$$
\begin{array}{ll}
[\![\, p \text{ when } g \,]\!]_\rho \;\; e \;\; s_1 \;\; s_2 = \{ & \text{let } \_valid \;=\; true; \\
 & [\![\, p \,]\!]_\rho \;\; e \; ([\![\, g \,]\!]_{\rho_1} \;\; \_x \\
 & (\text{if } (\_x) \; \{s_1\} \text{ else} \\
 & \{\_valid \;=\; false\})) \; (\_valid \;=\; false) \\
 & \text{if } (!\_valid)\{s_2\}\}
\end{array}
$$

Листинг 3.24: Правила трансляции сопоставления с образцом без повторения конструкции s_2

$$
\begin{array}{lll}
t & ::= & int \\
 & | & bool \\
 & | & string \\
 & | & t_1 * t_2 * \ldots * t_n \\
 & | & C\ t_1\ \ldots\ t_n \\
 & | & t \to t \\
CTor & ::= & \text{type } C\ x_1\ \ldots\ x_n\ =\ \{f_1 : t_1, \ldots, f_n : t_n\} \\
 & | & \text{type } C\ x_1\ \ldots\ x_n\ =\ t \\
 & | & \text{type } C\ x_1\ \ldots\ x_n\ = \\
 & & \quad | \ C_1 \text{ of } t_1 \\
 & & \quad \ldots \\
 & & \quad | \ C_n \text{ of } t_n \\
 & | & \text{type } C\ x_1\ \ldots\ x_n\ = \\
 & & \quad | \ C_1 \text{ of } t_1\ \to\ \ldots\ \to\ t_n
\end{array}
$$

Листинг 3.25: Синтаксис используемого в данной работе набор типов языка OCaml

$$
\begin{array}{lll}
t & ::= & number \\
 & | & bool \\
 & | & string \\
 & | & [t_1, t_2, \ldots, t_n] \\
 & | & C < t_1\ \ldots\ t_n > \\
 & | & t \mid t \\
 & | & t => t \\
CTor & ::= & \text{type } C < x_1\ \ldots\ x_n >=\ \{f_1 : t_1, \ldots, f_n : t_n\} \\
 & | & \text{type } C < x_1\ \ldots\ x_n >=\ t
\end{array}
$$

Листинг 3.26: Синтаксис используемого в данной работе набор типов языка JavaScript, предоставляемых инструментом Flow

$$
\begin{array}{ll}
[\![\ int\ ]\!]_\rho & = number \\
[\![\ bool\ ]\!]_\rho & = bool \\
[\![\ string\ ]\!]_\rho & = string \\
[\![\ (t_1 * t_2 * \ldots * t_n)\ ]\!]_\rho & = [[\![\ t_1\ ]\!]_\rho, [\![\ t_2\ ]\!]_\rho, \ldots, [\![\ t_n\ ]\!]_\rho] \\
[\![\ C\ t_1 \ldots t_n\ ]\!]_\rho & = C <[\![\ t_1\ ]\!]_\rho, [\![\ t_2\ ]\!]_\rho, \ldots, [\![\ t_n\ ]\!]_\rho> \\
[\![\ t_1 \to t_2\ ]\!]_\rho & = [\![\ t_1\ ]\!]_\rho => [\![\ t_2\ ]\!]_\rho \\
[\![\ \text{type } C\ x_1 \ldots x_n\ = & = \text{type } C < x_1 \ldots x_n > = \\
\ \{f_1 : t_1, \ldots, f_n : t_n\}\ ]\!]_\rho & \quad \{\_tag : \text{“}Record\text{“}, \\
 & \quad \_f_1 :[\![\ t_1\ ]\!]_\rho, \ldots, \_f_n :[\![\ t_n\ ]\!]_\rho\} \\
[\![\ \text{type } C\ x_1 \ldots x_n\ =\ t\ ]\!]_\rho & = \text{type } C < x_1 \ldots x_n > = [\![\ t\ ]\!]_\rho \\
[\![\ \text{type } C\ x_1 \ldots x_n\ = & = \text{type } C_1 < x_1 \ldots x_n > = \\
 & \quad \{\_tag : \text{“}C_1\text{“}, \_1 :[\![\ t_1\ ]\!]_\rho\} \\
\ | \ C_1 \text{ of } t_1 & \quad \ldots \\
\ldots & \quad \text{type } C_n < x_1 \ldots x_n > = \\
 & \quad \{\_tag : \text{“}C_n\text{“}, \_1 :[\![\ t_n\ ]\!]_\rho\} \\
\ | \ C_n \text{ of } t_n\ ]\!]_\rho & \quad \text{type } C < x_1 \ldots x_n > = \\
 & \quad C_1 < x_1 \ldots x_n > | \ldots | C_n < x_1 \ldots x_n > \\
[\![\ \text{type } C\ x_1 \ldots x_n\ = & = \text{type } C < x_1 \ldots x_n > = \\
| \ C_1 \text{ of } t_1\ \to\ \ldots\ \to\ t_n\ ]\!]_\rho & \quad \{\_tag : \text{“}C_1\text{“}, \_1 :[\![\ t_1\ ]\!]_\rho, \ldots, \_n :[\![\ t_n\ ]\!]_\rho\}
\end{array}
$$

Листинг 3.27: Правила трансляции OCaml-типов в Flow-типы

4 Архитектура инструмента и детали реализации

4.1 Архитектура инструмента для компиляции F*-программ в робастные Веб-приложения

В рамках проекта FStarLang [15] была создана и реализована архитектура инструмента для компиляции сертифицированных F*-программ в робастные Веб-приложения. В основе реализации инструмента лежат предложенные правила трансляции с языка F*, а именно, подмножества языка OCaml в

язык JavaScript. Диаграмма компонентов реализованного инструмента представлена на рис. 1. Желтым цветом выделена та часть, которая была сделана в рамках данной работы.

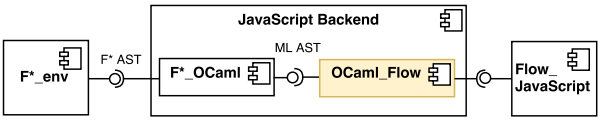


Рис. 1: Диаграмма компонентов реализованного инструмента

Ниже представлено описание каждого компонента.

Компонент *F*_env* отвечает за верификацию программы. Инструмент F* позволяет создавать программы и их верифицировать. Для того чтобы можно было их выполнять, необходим механизм извлечения верифицированного кода в программу на другом языке программирования. В проекте FStarLang такой механизм реализован для языка OCaml, схожий с [7]. Данный механизм удаляет зависимые и уточняющие типы, заменяя их стандартными типами целевого языка, ghost-вычисления и доказательства лемм, оставляя только их формулировки.

Компонент *JavaScript Backend* состоит из двух компонентов: *F*_OCaml* и *OCaml_Flow*. Компонент *F*_OCaml* отвечает за построение ML AST из F* AST, которое можно переиспользовать для создания новых бэкендов для инструмента F*. Ранее в проекте для старой версии инструмента использовался другой подход, а именно, трансляция языка F* в JavaScript напрямую [2]. Компонент *OCaml_Flow* отвечает за трансляцию ML AST в Flow AST с сохранением аннотаций типов. Результатом работы компонента *JavaScript Backend* является программа, которая может быть проверена инструментом Flow.

Компонент *Flow_JavaScript* необходим для получения JavaScript-приложения, которое может быть выполнено на программной платформе Node.js [13]. Данный компонент отвечает за удаление аннотаций типов и преобразование ES-стиля для работы с модулями в стиль CommandJS. Данное преобразование происходит с использованием соответствующих плагинов [12] и [11].

4.2 Процесс построения JavaScript-приложения из F*-программы

Пошаговый процесс получения JavaScript-приложения из F*-программы описан ниже (см. рис. 2).

- **Шаг 0:** На вход подается F*-программа, которая может состоять из нескольких модулей.
- **Шаг 1:** Для каждого модуля программы происходит построение F* AST.
- **Шаг 2:** Построение ML AST из F* AST путем удаления зависимых и уточняющих типов в F* AST, ghost-вычислений.
- **Шаг 3:** Трансляция ML AST в Flow AST с сохранением аннотаций для типов.
- **Шаг 4:** Получение программы, которая может быть проверена инструментом Flow. Количество модулей программы равняется количеству модулей исходной программы плюс количество модулей тех библиотек, чьи функции были использованы при создании программы.
- **Шаг 5:** Преобразование Flow-программы в JavaScript-программу (удаляется информация о типах, преобразование ES-стиль в CommandJS-стиль).

4.3 Работа с библиотечными функциями

Программа чаще всего состоит из нескольких модулей и использует библиотечные функции. В F* такие библиотеч-

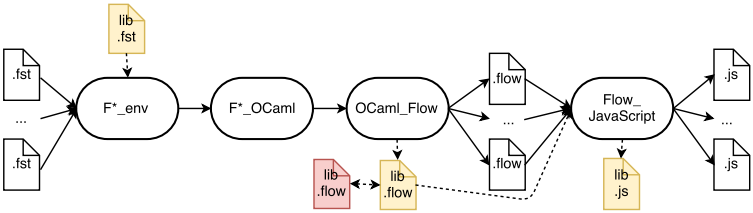


Рис. 2: Процесс построения JavaScript-приложения из F*-программы

ные функции уже проверифицированы, что упрощает доказательства многих программ. По умолчанию при трансляции F*-программ происходит трансляция каждого модуля и модулей тех библиотек, которые были использованы при создании программы. Однако реализация таких библиотек не предназначена для быстрого выполнения программы, поэтому разработчик может заменить ее на более быструю, использовав функции целевого языка. Ответственность за такую замену полностью лежит на разработчике. Для того чтобы избежать большего количества ошибок, данную замену необходимо осуществлять так, чтобы была возможность проверить, используя инструмент Flow, согласованность написанной библиотеки с извлеченной из F* программой, например, с точки зрения системы типов целевого языка. На рис. 2 библиотека, которая была получена в результате трансляции соответствующих модулей F*-библиотек, отмечена желтым цветом (и соответствующие файлы имеют расширение .flow). Красным цветом отмечена библиотека, которая содержит в себе эффективную реализацию функций, использующую возможности целевого языка.

4.4 Взаимодействие с модулями

Инструмент Flow поддерживает два стиля для работы с модулями, а именно, ES-стиль и CommandJS-стиль. В первом стиле используются конструкции *export/import*, а во втором — конструкции *require/exports*. При этом для типов осуществлена поддержка только ES-стиля, в то время как для переменных и функций поддержаны оба стиля ES и CommandJS. В данной работе для унификации трансляции использовался ES-стиль. Однако на данный момент он не поддерживается программной платформой Node.js [13], поэтому для последнего преобразования необходим плагин, который ES-стиль заменяет на CommandJS-стиль.

5 Экспериментальное исследование

Экспериментальное исследование организовано в соответствии с методом Goal Question Metric [6]. Этот метод описывает модель эксперимента, которая имеет иерархическую структуру и состоит из трех уровней. На первом уровне необходимо определить *цель* эксперимента, а также *тестовые данные* (evaluation objects), на которых будут проводиться эксперименты. На втором уровне необходимо сформулировать *вопросы*, которые помогут определить, достигнута ли поставленная цель или нет. На последнем уровне необходимо для каждого вопроса привести *метрики*, помогающие ответить на поставленные вопросы.

Целью данного исследования является проверка эффективности реализованного инструмента для компиляции верифицированных F*-программ в робастные Веб-приложения. В качестве evaluation object использовался проект HACL* [16]. Для достижения поставленной цели были сформулированы следующие *вопросы*.

Вопрос 1. Каково качество кодогенератора в JavaScript?

Вопрос 2. Какова эффективность результатов трансляции из F* в JavaScript?
Вопрос 3. Какова готовность инструмента для использования в индустрии?

Для ответа на первый вопрос были сформулированы следующие *метрики*.

M1.1. Количество верхнеуровневых функций в программе, которая получена в результате трансляции F*-программы в OCaml, и в программе, которая получена в результате трансляции F*-программы в JavaScript (Flow): F*_OCaml и F*_Flow.
M1.2. Количество строк кода: F*_OCaml и F*_Flow.

Для ответа на второй вопрос использовалась следующая *метрика*.

M2.1. Сравнение времени выполнения работы программ, одна из которых получена в результате трансляции F*-программы в JavaScript, а другая — сразу создавалась на языке JavaScript: F*_JavaScript и JavaScript.

Для ответа на последний вопрос были выбраны следующие *метрики*.

M3.1. Количество библиотечных функций F*, которые необходимо реализовать на JavaScript для полноценной работы оттранслированной программы.
M3.2. Конструкции языка F*, которые не поддержаны.

Данная апробация проходила на машине со следующими характеристиками и окружением:

- OS: Windows 10 Pro x64;
- Processor: Intel(R) Core(TM) i7-7500 CPU @ 2.70GHz;
- RAM: 16 Gb;

– OCaml 4.02.3, Cygwin_x86_64;
– Node.js v7.2.1, Flow 0.38.0.

Для ответа на первый вопрос использовалась реализация алгоритма Chacha20 [8] проекта HACL*. Для ее верификации использовалось около 37 библиотечных и вспомогательных модулей. Результаты измерений, сделанных согласно предложенным метрикам, приведены в таблице 1.

	F*	F*_OCaml	F*_Flow
Кол-во верхнеуровневых функций	32	32	32
Кол-во строк кода	786	422	316

Таблица 1: Результаты измерений, полученных по метрикам $M1.1$ и $M1.2$

Программы F*_OCaml и F*_Flow имеют меньшее количество строк кода, чем в исходной, потому что в ней использовалось около 9 лемм, которые при трансляции сохраняют только свою сигнатуру (см. листинги 3 и 4). При этом имена переменных исходной программы сохраняются в оттранслированных программах.

Для ответа на второй вопрос использовалась реализация алгоритма ChaCha20. Первая программа была получена в результате применения разработанного в рамках данной работы инструмента для компиляции F*-программы в JavaScript, где исходная программа была взята из репозитория проекта HACL*. Описательные характеристики ее приведены в первом вопросе. Вторая программа была взята из репозитория [18], в которой 257 строк кода. График сравнения времени выполнения программ JavaScript и F*_JavaScript приведен на рис. 3.

Выполнялось измерение времени работы функции *encrypt*, результатом которой является зашифрованный текст, полученный по исходному тексту произвольной длины, 256-bit ключу, 96-bit случайному коду (nonce) и 32-bit счетчику. Сообщения были получены с использованием функции

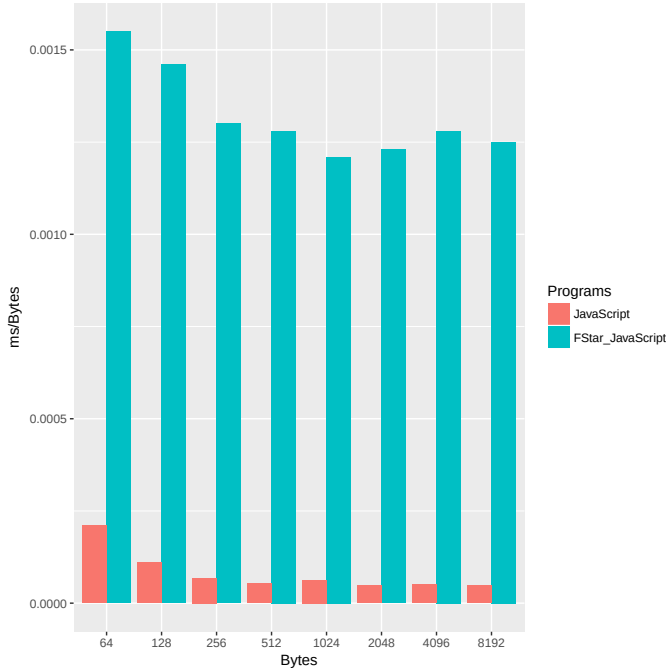


Рис. 3: График сравнения времени выполнения программ JavaScript и F*_JavaScript

crypto.randomBytes(*len*) из JavaScript-библиотеки *crypto*, где *len* – размер сообщения. Функция *encrypt* запускалась 3000 раз на одних и тех данных, после чего бралось среднее время выполнения этой функции в миллисекундах. На графике представлены измерения для сообщений, которые имели размер, соответственно, 64 байт, 128 байт, 256 байт, 512 байт, 1 КБайт, 2 КБайт, 4 КБайт и 8 КБайт. Значения даны в милли-

секундах на байт, то есть среднее время выполнения функции *encrypt* было разделено на размер сообщения.

Данный график показывает, что время выполнения работы программы JavaScript превосходит время выполнения работы программы F*_JavaScript. Данный факт объясняется тем, что в рамках данной работы выполнен только прототип инструмента, для которого в будущем будет проведен ряд оптимизаций. Например, замена оттранслированных функций на функции из целевого языка. С другой стороны, целью работы было создание безопасного и проверифицированного кода, что как известно, отражается на производительности программы. Одним из дальнейших направлений данной работы является реализация инструмента для компиляции F*-программы в JavaScript + WebAssembly [29]. Инструмент WebAssembly позволяет создавать быстрый и безопасный код, который можно использовать для совместной компиляции с JavaScript. При этом компиляция F*-кода в WebAssembly происходит с использованием KreMLin, который является промежуточным языком при трансляции подмножества языка F* в C.

Точного ответа дать на последний вопрос нельзя, так как язык F* активно развивается и разрабатывается. На текущий момент реализована только та часть библиотечных функций на JavaScript, которая активно использовалась в проводимых экспериментах и без которой нельзя было бы получить готовое приложение. Так как при трансляции программы происходит трансляция модулей библиотек, чьи функции использовались при создании и верификации программы, то работоспособность оттранслированных функций в JavaScript зависела от того, какую реализацию для них предоставил механизм извлечения F*-программы в OCaml. Конструкции языка F*, которые не были поддержаны в данной работы: исключения (Exceptions) и автогенерация конструкторов. Их поддержка будет добавлена при необходимости.

Заключение

При выполнении данной работы были получены следующие результаты:

- сформулированы правила трансляции с языка F* на JavaScript, гарантирующие сохранение аннотаций типов;
- выполнена реализация предложенного подхода на языке F*. Исходный код реализованного инструмента доступен по ссылке `https://github.com/FStarLang/FStar/tree/polubelova\_backends`, автор принимал участие под учетной записью polubelova;
- проведено экспериментальное исследование реализованного инструмента на примерах из криптографической библиотеки HACL*.

В дальнейшем планируется добавить возможность компиляции F*-программ в WebAssembly + JavaScript приложения и доказать корректность такой компиляции. Также планируется провести апробацию полученного инструмента на криптографической библиотеке HACL*, для которой нужно будет реализовать F*-библиотеки, используемые при верификации программ, на языке JavaScript.

Список литературы

1. Chlipala Adam. Certified programming with dependent types. — 2016.
2. Cédric Fournet Nikhil Swamy Juan Chen Pierre-Evariste Dagand Pierre-Yves Strub Ben Livshits. Fully Abstract Compilation to JavaScript // ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 2013. — ACM, 2013. — P. 371–384. — URL: `https://www.microsoft.com/en-us/research/publication/fully-abstract-compilation-to-javascript/`.
3. Dierks T., Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.2. — 2008. — URL: `https://tools.ietf.org/html/rfc5246`.

4. F* Tutorial. — URL: https://www.fstar-lang.org/tutorial/.
5. Gardner Philippa Anne, Maffeis Sergio, Smith Gareth David. Towards a Program Logic for JavaScript // Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '12. — ACM, 2012. — P. 31–44. — URL: http://doi.acm.org/10.1145/2103656.2103663.
6. Goal question metric (gqm) approach / Rini Van Solingen, Vic Basili, Gianluigi Caldiera, H Dieter Rombach // Encyclopedia of software engineering. — 2002.
7. Letouzey Pierre. Extraction in Coq: An Overview // Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings. — 2008. — P. 359–369. — URL: http://dx.doi.org/10.1007/978-3-540-69407-6_39.
8. Nir Y. ChaCha20 and Poly1305 for IETF Protocols. — 2015. — URL: https://tools.ietf.org/html/rfc7539.
9. Абстрактное синтаксическое дерево программ на Flow. — URL: https://github.com/facebook/flow/blob/master/src/parser/ast.ml.
10. Компиляторы. Принципы, технологии и инструментарий / А.В. Ахо, М.С. Лам, Р. Сети, Д. Д. Ульман. — Вильямс, 2016.
11. Плагин для преобразования export/import-стиля в exports/require-стиль. — URL: https://www.npmjs.com/package/babel-plugin-transform-flow-strip-types.
12. Плагин для удаления типов в JavaScript-программе. — URL: https://www.npmjs.com/package/babel-plugin-transform-flow-strip-types.
13. Платформа Node.js. — URL: https://nodejs.org/en/.
14. Рейтинг топ-10 языков программирования, используемых в веб-разработке. — URL: http://www.rswebsols.com/tutorials/programming/top-10-programming-languages-web-development.
15. Репозиторий проекта F*. — URL: https://github.com/FStarLang/FStar/.
16. Репозиторий проекта HACL*. — URL: https://github.com/mitls/hacl-star.
17. Репозиторий проекта KreMLin. — URL: https://github.com/FStarLang/kremlin.

18. Репозиторий проекта js-chacha20. — URL: https://github.com/thesimj/js-chacha20.
19. Сайт криптографической библиотеки OpenSSL. — URL: https://www.openssl.org/.
20. Сайт проекта Agda. — URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.
21. Сайт проекта BuckleScript. — URL: https://bloomberg.github.io/bucklescript/.
22. Сайт проекта Coq. — URL: https://coq.inria.fr/.
23. Сайт проекта Dafny. — URL: https://github.com/Microsoft/dafny.
24. Сайт проекта Everest. — URL: https://project-everest.github.io/.
25. Сайт проекта F*. — URL: https://www.fstar-lang.org/.
26. Сайт проекта Flow. — URL: https://flow.org/en/.
27. Сайт проекта Idris. — URL: http://www.idris-lang.org/.
28. Сайт проекта Vale. — URL: https://github.com/project-everest/vale.
29. Сайт проекта Web Assembly. — URL: http://webassembly.org/.
30. Сайт проекта Z3. — URL: http://z3.codeplex.com/.
31. Сайт проекта js_of_ocaml. — URL: http://ocsigen.org/js_of_ocaml/.
32. Сайт проекта miTLS. — URL: https://mitls.org/.
33. Спецификация языка ECMAScript 6. — URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm.
34. Список языков программирования, которые транслируются в JavaScript. — URL: https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js.
35. Сравнение инструмента Flow с языком TypeScript. — URL: http://djcordhose.github.io/flow-vs-typescript/.
36. Уязвимость Heartbleed. — URL: http://heartbleed.com/.
37. Язык программирования TypeScript. — URL: https://www.typescriptlang.org/.

Синтаксический анализ данных, представленных в виде контекстно-свободной грамматики

Ковалев Дмитрий Александрович

Санкт-Петербургский государственный университет
lares77@yandex.ru

Аннотация Многие программы в процессе работы формируют из строк исходный код на некотором языке программирования и передают его для исполнения в соответствующее окружение (пример — dynamic SQL). Для статической проверки корректности динамически формируемого выражения используются различные методы, одним из которых является синтаксический анализ регулярной аппроксимации множества значений такого выражения. Аппроксимация может содержать строки, не принадлежащие исходному множеству значений, в том числе синтаксически некорректные. Анализатор в данном случае сообщит об ошибках, которые на самом деле отсутствуют в выражении, генерируемом программой. В докладе будет описан алгоритм синтаксического анализа более точной, чем регулярная, контекстно-свободной аппроксимации динамически формируемого выражения.

Введение

Контекстно-свободные грамматики, наряду с регулярными выражениями, активно используются для решения задач, связанных с разработкой формальных языков и синтаксических

анализаторов. Одним из основных достоинств контекстно-свободных грамматик является возможность задания широкого класса языков при сохранении относительной компактности представления. Благодаря данному свойству, грамматики также представляют интерес в такой области информатики, как кодирование и сжатие данных. В частности, существует ряд алгоритмов, позволяющих производить сжатие текстовой информации, используя в качестве конечного [7] или промежуточного [2] представления контекстно-свободную грамматику (grammar-based compression).

Стандартной процедурой при работе с текстовыми данными является поиск в них определенных шаблонов, которые могут быть заданы строкой или регулярным выражением. В настоящее время большие объемы информации, как правило, хранятся и передаются по сети в сжатом виде, поэтому актуальной задачей становится поиск шаблонов непосредственно в компактном контекстно-свободном представлении текста. Такой подход позволяет избежать дополнительных затрат памяти на восстановление исходной формы данных и в некоторых случаях увеличивает скорость выполнения запроса. Шаблон здесь может быть, как и при поиске в обычном тексте, строкой (compressed pattern matching), сжатой строкой (fully compressed pattern matching) или регулярным выражением.

Известны ситуации, в которых для задания шаблона необходимо использовать более выразительные средства. Примером может служить одна из задач биоинформатики — поиск определенных подпоследовательностей в геноме организма. Так, для классификации и исследования образцов, полученных в результате процедуры секвенирования, в них могут искать гены, описывающие специфические рРНК. Структура таких генов, как правило, задается при помощи контекстно-свободной грамматики [8]. Для уменьшения объемов памяти, необходимых для хранения большого количества геномов, используются различные алгоритмы сжатия, в том числе осно-

ванные на получении контекстно-свободной структуры исходных последовательностей [3].

Задача поиска КС-шаблонов при использовании КС-представления данных формулируется следующим образом: необходимо найти все строки, принадлежащие пересечению двух языков, один из которых задается грамматикой шаблона, а второй представляет собой язык всех подстрок исходного множества строк, описываемого грамматикой, полученной в результате сжатия данных. Назовем такой поиск *синтаксическим анализом данных, представленных в виде КС-грамматики.* В общем случае задача неразрешима, так как сводится к задаче о проверке пересечения двух языков, порождаемых произвольными КС-грамматиками, на пустоту [5]. Для постановки экспериментов в области биоинформатики необходимо точнее исследовать возможность проведения синтаксического анализа КС-представления и разработать прототип алгоритма, позволяющего решить данную задачу.

1 Постановка задачи

Целью данной работы является разработка алгоритма синтаксического анализа данных, представленных в виде контекстно-свободной грамматики. Для ее достижения были поставлены следующие задачи.

- Определить ограничения, при которых синтаксический анализ контекстно-свободного представления является разрешимой задачей.
- Разработать алгоритм синтаксического анализа КС-представления данных с учетом поставленных ограничений.
- Реализовать предложенный алгоритм.
- Провести экспериментальное исследование.

2 Обзор

В данной работе используется понятие *рекурсивного автомата* [11] — удобного представления произвольной контекстно-свободной грамматики. Описание этой абстракции приводится в первом параграфе обзора.

Предлагаемый в работе алгоритм основан на алгоритме синтаксического анализа регулярных множеств, который, в свою очередь, является модификацией алгоритма обобщенного синтаксического анализа Generalized LL (GLL, [9]). Об этих алгоритмах и о проекте, в рамках которого проведена разработка предложенного решения, также будет рассказано в обзоре.

2.1 Рекурсивные автоматы и КС-грамматики

Введем понятие рекурсивного автомата, которое потребуется для дальнейшего изложения.

Определение 1. Рекурсивный автомат R — это кортеж $(\Sigma, Q, \delta, q_0, q_f)$, где Σ — конечное множество терминальных символов, Q — конечное множество состояний автомата, $\delta : Q \times (\Sigma \cup Q) \to 2^Q$ — функция переходов, $q_0 \in Q$ — начальное состояние, q_f — конечное состояние.

Можно заметить, что данное определение практически идентично определению стандартного конечного автомата. Единственное отличие состоит в том, что метками на ребрах рекурсивного автомата могут быть как терминальные символы (терминальные переходы), так и состояния (нетерминальные переходы). Класс рекурсивных автоматов обладает такой же выразительностью, как и контекстно-свободные грамматики, т.е. позволяет описать любой контекстно-свободный язык. Более того, грамматика тривиальным образом может быть преобразована в рекурсивный автомат (обратное тоже верно) [11]. Пример рекурсивного автомата, построенного по грамматике, можно увидеть на рис. 3.

$$S' ::= S$$
$$S ::= \texttt{[}\ S\ \texttt{]}$$
$$S ::= \texttt{a}$$

Рис. 1: Грамматика G_1

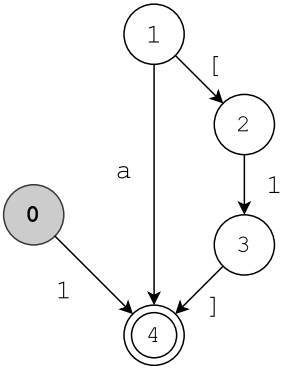


Рис. 2: Рекурсивный автомат для G_1

Рис. 3: КС-грамматика и эквивалентный ей рекурсивный автомат

2.2 GLL-алгоритм и его модификации

Классические алгоритмы нисходящего и восходящего синтаксического анализа предполагают использование грамматики, которая является в достаточной мере однозначной. В противном случае, управляющие таблицы анализаторов содержат конфликты, из-за чего нельзя гарантировать корректное поведение на любых входных данных. Для работы с сильно неоднозначными грамматиками используются алгоритмы *обобщенного синтаксического анализа*, которые позволяют рассмотреть все возможные пути разбора строки и построить соответ-

ствующие деревья вывода. Поиск шаблонов не требует наличия деревьев вывода, поэтому в дальнейшем алгоритмы синтаксического анализа рассматриваются только как механизм, позволяющий определить принадлежность строки языку.

Оригинальный GLL-алгоритм Generalized LL (GLL) — алгоритм, обобщающий идеи нисходящего синтаксического анализа. GLL, в отличие от стандартных алгоритмов LL-класса, позволяет использовать для анализа произвольную контекстно-свободную грамматику, в том числе содержащую леворекурсивные правила. Вместе с тем, GLL наследует такие полезные свойства алгоритмов нисходящего анализа, как непосредственная связь с грамматикой и простота отладки и диагностики ошибок.

Для обработки неоднозначностей GLL разделяет стек анализатора на несколько ветвей, каждая из которых соответствует возможному пути разбора. При таком подходе необходимо компактное представление множества стеков, в качестве которого выступает Graph Structured Stack (GSS). В работе [1] была представлена модификация GSS, которая позволяет увеличить эффективность GLL-анализа. Вершины такого представления хранят в себе номер нетерминала и позицию в строке, с которой начался разбор подстроки, соответствующей ему. На ребрах хранятся позиции в грамматике (вида $X \to \alpha A \cdot \beta$), на которые необходимо вернуться после завершения разбора нетерминала.

Основной идеей GLL является использование *дескрипторов*, позволяющих полностью описывать состояние анализатора в текущий момент времени.

Определение 2. Дескриптор — это тройка (L, u, i), где:

- L — текущая позиция в грамматике вида $A \to \alpha \cdot \beta$;
- u — текущая вершина GSS;
- i — позиция во входном потоке.

В процессе работы поддерживается глобальная очередь дескрипторов. В начале каждого шага исполнения алгоритм берет следующий в очереди дескриптор и производит действия в зависимости от позиции в грамматике и текущего входного символа, передвигая соответствующие указатели. При наличии конфликтов в грамматике алгоритм добавляет дескрипторы для каждого возможного пути анализа в конец очереди.

Поддержка грамматик в EBNF В работе [4] была описана модификация GLL, которая позволяет использовать грамматики, записанные в расширенной форме Бэкуса-Наура (EBNF). Грамматика такого вида трансформируется в соответствующий рекурсивный автомат, в котором затем минимизируется количество состояний. Синтаксический анализ производится без построения управляющих таблиц: алгоритм обходит рекурсивный автомат в соответствии со входным потоком символов. При обработке текущего дескриптора (C_S, C_U, i), где C_S — вершина автомата (эквивалент позиции в грамматике), C_U — вершина GSS, i — позиция в строке, могут возникать следующие ситуации.

- C_S — финальное состояние. Показывает, что разбор текущего нетерминала завершен. Необходимо осуществить возврат из C_U по меткам на исходящих из нее ребрах.
- Присутствует нетерминальный переход из C_S. В данном случае необходимо начать разбор указанного нетерминала X. Для этого в GSS должна быть создана новая вершина (X, i), если таковой не было ранее, а текущая вершина автомата изменена на стартовую для X.
- Присутствует терминальный переход из C_S. Необходимо сравнить терминал на ребре автомата с текущим входным символом. Если они совпадают, то осуществить переход в вершину автомата, на которую указывает ребро, и передвинуть указатель в строке.

За счет уменьшения количества состояний в автомате удается достичь прироста в производительности по сравнению со стандартным GLL-алгоритмом.

Синтаксический анализ графов Стандартными входными данными для алгоритмов синтаксического анализа являются линейные последовательности токенов. На основе GLL был разработан алгоритм, который позволяет производить синтаксический анализ регулярных множеств строк, представленных в виде конечного автомата (который, в свою очередь, является ориентированным графом с токенами на ребрах).

Поддержка нелинейного входа не потребовала существенных изменений в оригинальном алгоритме. Дескрипторы модифицированного алгоритма хранят номер вершины входного графа вместо позиции в строке. Также, на шаге исполнения просматривается не единственный текущий символ, а множество символов на ребрах, исходящих из текущей вершины.

Производительность данного алгоритма, как и обычного GLL, может быть увеличена при помощи представления входной грамматики в виде рекурсивного автомата. В таком случае, алгоритм будет производить обход двух автоматов — рекурсивного и конечного. Ситуации, возникающие при обработке дескрипторов, не отличаются от описанных ранее ситуаций для линейного входа. Псевдокод данной модификации приведен в приложении. Рассматривается вариант без построения деревьев вывода, алгоритм возвращает длины корректных цепочек, порождаемых автоматом.

2.3 Проект YaccConstructor

YaccConstructor [13] — исследовательский проект лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ, направленный на исследования в области лексического и синтаксического анализа. Проект включает в себя одноименную модульную платформу

для разработки лексических и синтаксических анализаторов, содержащую большое количество компонент: язык описания грамматик YARD [12], преобразования над грамматиками и др. Основным языком разработки является F#.

Ранее в рамках YaccConstructor были реализованы генераторы GLL-анализаторов, описание которых было приведено в данном обзоре.

3 Разрешимость задачи синтаксического анализа контекстно-свободного представления

Как было сказано ранее, задачу поиска шаблона, при условии, что и шаблон, и данные, в которых осуществляется поиск, представлены контекстно-свободными грамматиками, мы назовем синтаксическим анализом контекстно-свободного представления. В данном разделе определяются ограничения, при которых подобная задача разрешима.

Для доказательства предложений, сформулированных далее, используется следующая теорема [6].

Theorem 1 (Nederhof, Satta). *Пусть G_1 — произвольная контекстно-свободная грамматика, G_2 — грамматика, которая не содержит непосредственной или скрытой рекурсии. Тогда проблема проверки пустоты пересечения языков, порождаемых данными грамматиками, относится к классу PSPACE-complete.*

Рассмотрим случай, когда грамматика данных задает ровно одну строку. Пусть G_t — произвольная КС-грамматика, задающая шаблоны для поиска, а G_d — КС-грамматика, которая не содержит непосредственной или скрытой рекурсии. $L(G_t)$ и $L(G_d)$ — языки, порождаемые грамматиками, при этом $L(G_d) = \{\omega\}$, где ω — исходные данные, к которым

был применен алгоритм сжатия. Необходимо определить, существуют ли такие строки ω', что $\omega' \in L(G_t)$ и ω' — подстрока ω.

Предложение 1. *При выполнении описанных условий задача синтаксического анализа KC-представления разрешима.*

Доказательство. Пользуясь эквивалентностью представлений, можно записать грамматику G_d в виде рекурсивного автомата R_d. Рассмотрим рекурсивный автомат $R_{i,\,j}$, полученный из R_d путем замены стартового состояния на $i \in Q(R_d)$ и назначения терминирующего (финального, из которого не могут быть совершены переходы) состояния $j \in Q(R_d)$. Такой автомат описывает грамматику, которая является представлением некоторой подстроки ω. Рассмотрев все возможные пары i и j, получаем конечное множество грамматик, для каждой из которых необходимо проверить, содержится ли строка, порождаемая ей, в языке $L(G_t)$. Согласно теореме 1, такая проверка является разрешимой задачей и принадлежит к классу PSPACE-complete. □

Отдельно отметим, что для описанных процедур используется лишь исходный автомат, эквивалентный грамматике G_d. Условия задачи поиска шаблонов непосредственно в контекстно-свободном представлении, таким образом, выполняются. Верна также разрешимость более общей задачи.

Предложение 2. *Пусть грамматика G_d задает конечное множество строк $L(G_d) = \{\omega_1, \ldots, \omega_n\}$. Необходимо определить, существуют ли строки ω', для которых верно: $\omega' \in L(G_t)$ и ω' — подстрока одной из строк $\omega_i \in L(G_d)$. Данная задача разрешима и принадлежит классу PSPACE-complete.*

Доказательство. Как и в предыдущем доказательстве, используем запись грамматики в виде рекурсивного автомата R_d и рассмотрим автоматы $R_{i,j}$. В данном случае каждый из этих автоматов представляет собой грамматику, которая порождает некоторое конечное множество подстрок исходных строк из

$L(G_d)$. Проверка пустоты пересечения такой грамматики с G_t также соответствует условиям теоремы 1. □

В случае, когда грамматика G_d представляет собой бесконечный регулярный язык (т.е. содержит левую и/или правую рекурсию), разрешимость задачи поиска шаблонов установить не удается. Подход, использованный ранее в доказательстве предложений, не может быть применен, так как части рекурсивного автомата, представляющего грамматику G_d, также могут содержать рекурсивные переходы, что выходит за рамки условия теоремы 1. Проверка разрешимости и определение класса сложности задачи проверки пустоты пересечения произвольной и регулярной КС-грамматик в настоящее время остаются открытыми проблемами [6].

4 Алгоритм синтаксического анализа контекстно-свободного представления

Разработанный алгоритм расширяет подход к синтаксическому анализу графов на основе GLL. На вход алгоритм принимает два рекурсивных автомата, RA_1 и RA_2, построенных по исходным грамматикам G_1 и G_2 соответственно, где G_1 — произвольная контекстно-свободная грамматика, а G_2, исходя из ограничений разрешимости задачи, — грамматика, не содержащая непосредственной или скрытой рекурсии.

Алгоритм производит обход автоматов, рассматривая три типа ситуаций (финальное состояние, терминальный/нетерминальный переходы) для каждого из их состояний. Автомат, состояние которого обрабатывается в данный момент, будем называть *основным*; другой автомат из пары назовем *побочным*.

Оригинальный алгоритм синтаксического анализа графов использует один GSS, необходимый для правильного обхода рекурсивного автомата, который является представлением эталонной грамматики. Для работы с двумя рекурсивными

автоматами требуется поддерживать одновременно два стека, GSS_1 и GSS_2. При этом используется модификация GSS с измененной структурой вершин: здесь они представляют собой тройку (S, i, u), где:

- S — нетерминал;
- i — состояние побочного автомата;
- u — вершина стека побочного автомата.

Пара (i, u) описывает состояние обхода побочного автомата на момент начала разбора нетерминала S. В оригинальном алгоритме достаточно было сохранять в вершине GSS лишь состояние входного автомата, так как подразумевалось, что даный автомат относится к классу конечных и для его обхода не используется стек.

Дескрипторы, которые использует разработанный алгоритм, также отличаются от представленных ранее. Для описания процесса анализа в определенный момент времени теперь необходимо указывать вершины GSS для обоих автоматов. Получаем дескриптор вида (S_1, S_2, u_1, u_2), где S_1, S_2 — состояния автоматов, а u_1, u_2 — вершины соответствующих стеков. Функции **add**, **create** и **pop** были изменены в соответствии с новыми определениями дескрипторов и вершин GSS.

Algorithm 1 Функции для работы с GSS и дескрипторами

```
function ADD(S_1, S_2, u_1, u_2)
  if ((S_1, S_2, u_1, u_2) ∉ U) then
    U.add(S_1, S_2, u_1, u_2)
    R.enqueue(S_1, S_2, u_1, u_2)
function CREATE(S_call, S_next, u, S_i, w)
  A ← Δ(S_call)
  if (∃ GSS node labeled (A, i, w)) then
    v ← GSS node labeled (A, i)
    if (there is no GSS edge from v to u labeled S_next) then
      add GSS edge from v to u labeled S_next
      for ((v, j) ∈ P) do
        if (S_next is a final state) then
          POP(u, j, w)
        ADD(S_next, S_i, u, w)
  else
    v ← new GSS node labeled (A, i, w)
    create GSS edge from v to u labeled S_next
    ADD(S_call, S_i, v, w)
function POP(u, S_i, w)
  if ((u, i) ∉ P) then
    P.add(u, i)
    for all GSS edges (u, S, v) do
      if (S is a final state) then
        POP(v, S_i, w)
      ADD(S, S_i, v, w)
```

Обработка терминальных и нетерминальных переходов, а также финальных состояний автомата была вынесена из главного цикла алгоритма в отдельную функцию, которая вызывается для каждого из автоматов.

Algorithm 2 Функция для обработки состояния рекурсивного автомата

```
function HANDLESTATE(RA, context)
  S_1, S_2, u_1, u_2 ← get values for RA from the context
  if (S_1 is a final state) then
    POP(u_1, S_2, u_2)
  for each transition(S_1, label, S_next) do
    switch label do
      case Terminal(x)
        for each (input[i] -x-> input[k]) do
          if (S_next is a final state) then
            POP(u_1, k, u_2)
          if S_next have multiple ingoing transitions then
            ADD(S_next, k, u_1, u_2)
          else
            R.enqueue(S_next, k, u_1, u_2)
      case Nonterminal(S_call)
        CREATE(S_call, S_next, u1, S_2, u_2)
```

Основная управляющая функция алгоритма, содержащая цикл обработки дескрипторов, представлена в листинге 3. Необходимо отметить, что для поиска шаблонов-подстрок обход автомата RA_2 запускается из каждой его вершины. Соответствующие стартовые дескрипторы создаются перед входом в цикл.

Algorithm 3 Алгоритм синтаксического анализа КС-представления

function PARSE(RA_1, RA_2)
  for each $q \in RA_2.States$ **do**
    $v1, v2 \leftarrow (RA_1.StartState, q, \$), (q, RA_1.StartState, \$)$
    add $v1$ to GSS_1, $v2$ to GSS_2
    $R.enqueue(RA_1.StartState, q, v1, v2)$
  while $R \neq \emptyset$ **do**
    $currentContext \leftarrow R.dequeue()$
    HANDLESTATE($RA_1, currentContext$)
    HANDLESTATE($RA_2, currentContext$)
  $result \leftarrow \emptyset$
  for each $(u, i) \in P$ where $u = GSS_1.Root$ **do**
    $result.add(i)$
  if $result \neq \emptyset$ **then** return $result$
  else report failure

Результатом работы алгоритма являются пары вида (n_1, n_2), где n_1, n_2 — номера состояний автомата RA_2. Для каждой из таких пар выполняется следующее утверждение: $\exists \omega \in T^*$ такая, что $\omega \in L(G_1)$ и $\omega \in L(RA')$, где RA' — рекурсивный автомат, полученный из RA_2 путем замены начального и конечного состояний на n_1 и n_2 соответственно.

5 Экспериментальное исследование

Предложенный алгоритм был реализован на платформе .NET как часть исследовательского проекта YaccConstructor. Основным языком разработки являлся F#. Был переиспользован генератор рекурсивных автоматов, реализованный ранее в рамках работы [4]. Также были переиспользованы структуры данных для представления структурированного в виде графа стека.

Для проверки работоспособности и оценки производительности алгоритма был проведен ряд тестов. В качестве представления шаблонов для поиска была выбрана грамматика G_1, задающая язык правильных скобочных последовательностей. Входные данные, в которых осуществлялся поиск, создавались следующим образом.

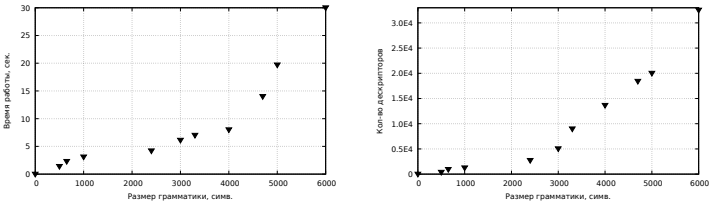
1. Из символов латинского алфавита генерировалась последовательность определенной длины, содержащая ровно пять подстрок, удовлетворяющих грамматике G_1.
2. Последовательность сжималась в контекстно-свободную грамматику при помощи алгоритма Sequitur [7, 10].
3. Описание полученной грамматики транслировалось в язык YARD, используемый в проекте YaccConstructor.

В каждом тесте из серии алгоритм корректно определял наличие шаблонов, получая пять пар вершин рекурсивного автомата, построенного по входной грамматике.

Оценка производительности заключалась в измерении времени работы алгоритма и количества создаваемых дескрипторов (рис. 4), а также объема используемой памяти, выраженного в количестве вершин и ребер GSS, построенных в процессе анализа (рис. 5). Тестирование проводилось при различных размерах входной грамматики данных. Размером грамматики $G = (\Sigma, N, P, S)$ назовем следующую величину:

$$|G| = |N| + \sum_{p \in P} length(p)$$

Для тестов использовался ПК со следующими характеристиками:
MS Windows 10 x64, Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz 4 Cores, 16GB.



(a) Время работы (b) Кол-во дескрипторов

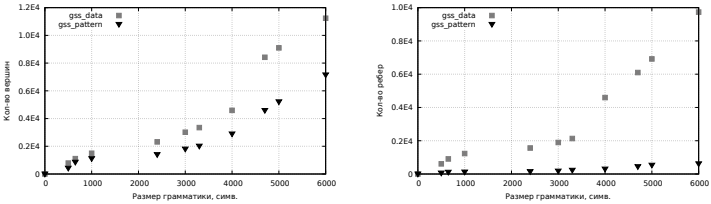
Рис. 4: Зависимость времени работы и кол-ва дескрипторов от размера входной грамматики

Результаты тестирования показывают, что даже в случае поиска простого шаблона время работы алгоритма быстро возрастает при увеличении размеров грамматики, представляющей данные. Предположительно, производительность может быть улучшена путем технической доработки используемых структур данных и самого алгоритма. Помимо этого, влияние на работу алгоритма может оказывать не только размер входной грамматики, но и ее структура, которая зависит от используемого алгоритма сжатия.

Заключение

В ходе данной работы получены следующие результаты.

– Определены ограничения, при которых синтаксический анализ



(a) Кол-во вершин (b) Кол-во ребер

Рис. 5: Зависимость размера построенных GSS от размера входной грамматики. Для обозначения GSS, соответствующего грамматике G_1, используется метка *gss_pattern*, входной грамматике — *gss_data*

контекстно-свободного представления является разрешимой задачей. Было показано, что грамматика, являющаяся представлением данных, должна порождать конечный язык.

– Разработан алгоритм синтаксического анализа КС-представления, учитывающий поставленные ограничения. Полученный алгоритм является модификацией алгоритма синтаксического анализа графов на основе GLL.
– Выполнена реализация алгоритма на языке программирования F# в рамках исследовательского проекта YaccConstructor.
– Проведено экспериментальное исследование: выполнено тестирование производительности на синтетических данных.

В дальнейшем планируется провести апробацию алгоритма на реальных данных — сжатом представлении ДНК организмов — и доказать теоретическую оценку сложности алгоритма.

Исходный код проекта YaccConstructor представлен на сайте
`https://github.com/YaccConstructor/YaccConstructor`.

Список литературы

1. Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings / Ed. by Björn Franke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. — P. 89–108. — ISBN: 978-3-662-46663-6. — URL: http://dx.doi.org/10.1007/978-3-662-46663-6_5.
2. Arimura Mitsuharu. A grammar-based compression using a variation of Chomsky normal form of context free grammar // 2016 International Symposium on Information Theory and Its Applications (ISITA). — 2016.
3. Gallé Matthias. Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem : Theses / Matthias Gallé ; Université Rennes 1. — 2011. — Feb. — URL: https://tel.archives-ouvertes.fr/tel-00595494.
4. Gorokhov Artem, Grigorev Semyon. Extended Context-Free Grammars Parsing with Generalized LL. — 2017.
5. Harrison M. A. Introduction to Formal Language Theory. — 1st edition. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1978. — ISBN: 0201029553.
6. Nederhof Mark-Jan, Satta Giorgio. The Language Intersection Problem for Non-recursive Context-free Grammars // Inf. Comput. — 2004. — Aug. — Vol. 192, no. 2. — P. 172–184. — URL: http://dx.doi.org/10.1016/j.ic.2004.03.004.
7. Nevill-Manning Craig G., Witten Ian H. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm // J. Artif. Int. Res. — 1997. — Sep. — Vol. 7, no. 1. — P. 67–82. — URL: http://dl.acm.org/citation.cfm?id=1622776.1622780.
8. Quantifying variances in comparative RNA secondary structure prediction / James WJ Anderson, Ádám Novák, Zsuzsanna Sükösd et al. // BMC Bioinformatics. — 2013. — Vol. 14, no. 1. — P. 149. — URL: http://dx.doi.org/10.1186/1471-2105-14-149.
9. Scott Elizabeth, Johnstone Adrian. GLL parsing // Electronic Notes in Theoretical Computer Science. — 2009. — Vol. 253, no. 7.
10. Sequitur [Электронный ресурс]. — URL: https://github.com/craignm/sequitur (online; accessed: 25.05.2017).

11. Tellier Isabelle. Learning recursive automata from positive examples // Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle. — 2006. — Vol. 20, no. 6. — P. 775–804.
12. YARD [Электронный ресурс]. — URL: `http://yaccconstructor.github.io/YaccConstructor/yard.html` (online; accessed: 25.05.2017).
13. Григорьев С. В. Синтаксический анализ динамически формируемых программ : Дисс... кандидата наук / С. В. Григорьев ; Санкт-Петербургский государственный университет. — 2015.

Приложение

Псевдокод алгоритма синтаксического анализа графов

Пусть (C_S, C_U, i, l) — текущий дескриптор, где C_S — состояние рекурсивного автомата, представляющего грамматику, C_U — вершина GSS, i — вершина входного графа, l — длина разобранной части строки. Для получения имени нетерминала грамматики, соответствующего состоянию автомата используется функция $\Delta : Q \to N$.

Во время работы алгоритма поддерживаются следующие множества: R — глобальная очередь дескрипторов, U — множество созданных ранее дескрипторов, P — множество, хранящее информацию о вызовах функции **pop**.

function ADD(S, u, i, l)
  if $((S, u, i, l) \notin U)$ **then**
    $U.add(S, u, i, l)$
    $R.enqueue(S, u, i, l)$

function CREATE$(S_{call}, S_{next}, u, i, l)$
  $A \leftarrow \Delta(S_{call})$
  if ($\exists$ GSS node labeled (A, i)) **then**
    $v \leftarrow$ GSS node labeled (A, i)

```
      if (there is no GSS edge from v to u labeled (S_next, l)) then
        add GSS edge from v to u labeled (S_next, l)
        for ((v, j, m) ∈ P) do
          if (S_next is a final state) then
            POP(u, j, (l + m))
          ADD(S_next, u, j, (l + m))
    else
      v ← new GSS node labeled (A, i)
      create GSS edge from v to u labeled (S_next, l)
      ADD(S_call, v, i, 0)

function POP(u, i, l)
  if ((u, i, l) ∉ P) then
    P.add(u, i, l)
    for all GSS edges (u, S, m, v) do
      if (S is a final state) then
        POP(v, i, (l + m))
      ADD(S, v, i, (l + m))

function PARSE(RA, input)

GSSroot ← newGSSnode(RA.StartState, input.StartState)
  R.enqueue(RA.StartState, GSSroot, input.StartState, 0)
  while R ≠ ∅ do
    (C_S, C_U, i, l) ← R.dequeue()
    if (l = 0) and (C_S is a final state) then
      POP(C_U, i, 0)
    for each transition(C_S, label, S_next) do
      switch label do
        case Terminal(x)
          for each (input[i] -x-> input[k]) do
            if (S_next is a final state) then
              POP(C_U, k, (l + 1))
            if S_next have multiple ingoing transitions then
              ADD(S_next, C_U, k, (l + 1))
```

```
                else
                    R.enqueue(S_next, C_U, k, (l + 1))
            case Nonterminal(S_call)
                CREATE(S_call, S_next, C_U, i, l)
    result ← ∅
    for each (u, i, l) ∈ P where u = GSSroot, i = input.FinalState do
        result.add(l)
    if result ≠ ∅ then return result
    else report failure
```

Разработка механизма использования OpenCL-кода в программах на F#

Смиренко Кирилл Петрович

Санкт-Петербургский государственный университет
k.smirenko@gmail.com

Введение

Графические процессоры (GPU) являются общепринятым средством ускорения вычислений. Многоядерная архитектура видеопроцессоров даёт преимущество в высоконагруженных научных вычислениях, задачах компьютерного зрения, биоинформатики и других областей. Данный подход лёг в основу техники вычислений общего назначения на видеопроцессорах (GPGPU) [5, 7].

Существует ряд технологий для программирования видеопроцессоров. Наиболее распространённой является CUDA — платформа для параллельных вычислений на видеопроцессорах, разработанная компанией Nvidia в 2007 году [14]. Другим значимым проектом является Open Computing Language (OpenCL) — открытый стандарт кросс-платформенного, параллельного программирования различных процессоров, в том числе видопроцессоров, присутствующих в персональных компьютерах, серверах, мобильных и встраиваемых устройствах [8].

Упомянутые технологии предоставляют специальные языки программирования: CUDA C/C++, OpenCL C/C++. В то же время более удобным способом разработки для видеопроцессоров является использование более высокоуровневых языков, таких, как C# и F#. Эти языки чаще применяются для

написания конечных приложений; кроме того, строгая типизация и статический анализ в интегрированных средах разработки повышают удобство прикладного программирования и надёжность разрабатываемого кода. Уже существует и активно используется ряд средств для высокоуровневого программирования видеопроцессоров [2, 4, 17].

При этом возникает потребность уметь переиспользовать в языках высокого уровня существующий код на специальных языках для видеопроцессоров. Это продиктовано следующими соображениями. Во-первых, переиспользование готового и проверенного кода вместо переписывания — стандартная инженерная практика. Во-вторых, низкоуровневый код для видеопроцессоров содержит специфические конструкции, например, барьеры и модификаторы памяти, часто используемые для оптимизаций. Пытаться выразить эти конструкции в языка высокого уровня лишь затем, чтобы потом транслировать их обратно в код целевой платформы (CUDA, OpenCL) для запуска на видеопроцессоре, не представляется рациональным решением.

Таким образом, является актуальной задача переиспользования низкоуровневого кода для GPU в высокоуровневом программировании. При этом отдельный интерес представляет возможность вызова низкоуровневых функций типизированным образом, что существенно повысило бы удобство прикладного программирования.

A Обзор

A.1 Средства программирования видеопроцессоров

Существует ряд инструментов, предназначенных для программирования видеопроцессоров на языках программной платформы .NET. Наиболее известные из них представлены ниже.

- Alea GPU — коммерческий продукт от компании QuantAlea, предоставляющий средства разработки для платформы CUDA на языках C# и F# [17].
- Brahma.FSharp — проект с открытым исходным кодом, разрабатываемый на кафедре системного программирования математико-механического факультета СПбГУ. Это транслятор цитируемых выражений языка F# в код платформы OpenCL [2].
- FSCL — другой компилятор F#-кода в OpenCL C с открытым исходным кодом [4].

Также существуют программные проекты, позволяющие из C#-кода управлять запуском кода на специальных языках для видеопроцессора. К ним относятся:

- Alea GPU, предоставляющий возможность вызова ряда готовых низкоуровневых библиотек [17];
- CUSP — C++-библиотека для вычислений с разреженными матрицами и обработки графов [3];
- ManagedCUDA — проект, содержащий средства нетипизированного вызова скомпилированных CUDA-функций в коде на C# и C#-интерфейс для ряда наиболее популярных CUDA-библиотек. [11].

Однако все имеющиеся решения либо только транслируют высокоуровневый код на языках платформы .NET в низкоуровневый код платформ CUDA или OpenCL (Brahma.FSharp, FSCL), либо позволяют использовать лишь фиксированный набор низкоуровневых библиотек (Alea GPU, CUSP). ManagedCUDA является наиболее близким решением, однако не предоставляет типизированных функций вызова низкоуровневого кода и не является транслятором .NET-кода в код платформы CUDA, позволяя работать лишь с предварительно скомпилированной кодовой базой.

В результате обзора можно сделать вывод, что ни одно из существующих решений не позволяет программировать видеопроцессоры на языке высокого уровня и при этом вызывать произвольный низкоуровневый код типизированным образом.

A.2 Провайдеры типов языка F#

Существует несколько способов интеграции низкоуровневого кода и среды исполнения .NET. Особый интерес представляют провайдеры типов F# [13]. Это механизм языка, который генерирует типы данных и встраивает в окружение времени исполнения как обычные типы F#. Провайдеры типов могут иметь типовые параметры, и, таким образом, имеется возможность во время разработки на F# пользоваться статической типизацией данных, которые были получены из динамических источников, например, из файла.

Традиционной и наиболее близкой альтернативой использованию провайдеров типов является кодогенерация. Однако по сравнению с ней провайдеры обладают рядом преимуществ:

- провайдеры типов обеспечивают тесную интеграцию с пользовательским контекстом: сгенерированные типы сразу находятся в одном пространстве имён с кодом, который их использует;
- генерация типов происходит во время компиляции пользовательского кода, что избавляет от опасности рассинхронизации с источником данных.

Провайдеры типов не лишены недостатков. В частности, затруднено тестирование проекта, в котором есть провайдеры типов: тесты, как и любой код, использующий провайдеры, не могут быть включены в ту же сборку, что и провайдер, так как блокируют пересборку последнего. Кроме того, отладка провайдеров типов представляет собой достаточно сложную процедуру, отличающуюся от процесса отладки обычного кода [18]. Тем не менее, данные недостатки относятся к процессу разработки программного решения, но не затрудняют его использование.

Таким образом, именно механизм провайдеров типов F# был выбран для интеграции разобранного OpenCL-кода и платформы .NET.

B Постановка задачи

Целью данной работы является добавление возможности переиспользования OpenCL C-кода в проект Brahma.FSharp. Для её достижения были поставлены следующие задачи:

- исследовать возможности механизма провайдеров типов F#;
- реализовать лексический и синтаксический анализатор заголовков OpenCL-функций;
- обеспечить возможность типизированного вызова OpenCL-функций в коде на F#;
- провести экспериментальное исследование представленного решения.

C Механизм подгрузки OpenCL-кода

C.1 Архитектура

На схеме 1 представлена архитектура предлагаемого решения. Подгружаемый OpenCL C-файл подаётся последовательно лексическому, синтаксическому анализатору и провайдеру типов. На выходе генерируется F#-тип, содержащий подгруженные функции. Эти функции могут быть использованы в клиентском коде — цитируемом выражении F#, которое будет подано существующему транслятору Brahma.FSharp, а результат трансляции — передан драйверу OpenCL для запуска на видеопроцессоре.

Реализуемый в рамках настоящей работы модуль, таким образом, можно разделить на три составляющих: лексический анализатор, синтаксический анализатор, провайдер типов. Далее будет произведён их подробный обзор.

C.2 Лексический и синтаксический анализатор

Для того, чтобы извлекать и переиспользовать сигнатуры функций OpenCL C, нужно уметь разбирать исходный код,

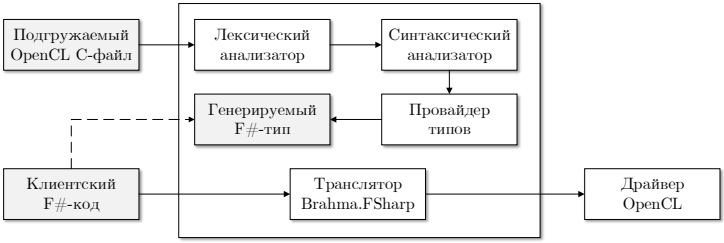


Рис. 1: Структура предлагаемого решения

т.е. проводить лексический и синтаксический анализ. Для реализации лексического анализатора был использован FsLex — инструмент с открытым исходным кодом, генерирующий исходный код лексических анализаторов на языке F# по данному описанию языка в специальной нотации [6].

При написании синтаксического анализатора была использована библиотека YaccConstructor [20, 22]. Она предоставляет язык YARD [21] для описания синтаксических анализаторов. Этот язык имеет ряд преимуществ перед аналогами [1,10] применимо к данной задаче: естественная интеграция с F#, возможность описывать грамматики в расширенной форме Бэкуса-Наура [19], поддержка S- и L-атрибутов, а также параметризованных правил вывода.

Синтаксический анализатор описан в виде S-атрибутной грамматики в расширенной форме Бэкуса-Наура. В качестве справочных материалов была использована формальная грамматика C99 [16] и спецификация языка OpenCL C [9]. Исходный код обоих анализаторов на F# генерируется упомянутыми инструментами при сборке проекта.

Представленная грамматика является упрощённой и не описывает весь язык OpenCL C: тот факт, что в данной задаче необходимо распознавать лишь заголовки функций, позволил при разборе игнорировать тела функций.

C.3 Провайдер типов для OpenCL-функций

В рамках работы был реализован провайдер типов для разобранных OpenCL-функций. Провайдер генерирует F#-тип, содержащий функции, которые типизированы так же, как исходные функции на OpenCL C. Провайдер параметризован путём к файлу, содержащему подгружаемый код на OpenCL C; файл может содержать неограниченное количество функций, однострочные комментарии и макросы препроцессора C. Вначале файл с исходным кодом на OpenCL C подвергается лексическому и синтаксическому анализу. Далее производится обход синтаксического дерева, полученного на этапе синтаксического анализа: из дерева извлекаются структурированные сигнатуры функций, по которым провайдер генерирует статические методы предоставляемого метода.

В C-подобных языках, включая OpenCL C, массивы чаще всего передаются по указателям. Поэтому для параметров функций, которые являются указателями, провайдер поддерживает два варианта отображения их в F#: как ссылочный тип и как массив. Это задаётся другим параметром провайдера. Пример подгрузки с помощью провайдера функции умножения матрицы на вектор представлен на изображении 2.

```
let matvec = KernelProvider<matvecPath, TreatPointersAsArrays=true>.matvec
```

Рис. 2: Пример использования провайдера типов

Также была проведена точечная доработка транслятора Brahma.FSharp. Транслятор одновременно с клиентским F#-кодом получает имя подгружаемого OpenCL C-файла, читает его содержимое и в текстовом виде передаёт драйверу OpenCL вместе с результатом трансляции F#-кода. Это обеспечивает необходимую функциональность.

D Эксперименты

Помимо модульного тестирования представленного механизма, были произведены экспериментальные исследования работы механизма с высокопроизводительным кодом для видеопроцессора. Это отвечает изначальной мотивации данной работы. В целях эксперимента из стороннего проекта с открытым исходным кодом [12] был взят алгоритм перемножения вещественных матриц на видеопроцессоре, оптимизированный с использованием специфических конструкций OpenCL: барьеров и локальных групп.

Для сравнения производительности были также взяты неоптимизированные реализации наивного алгоритма перемножения матриц на F# и OpenCL C. Оптимизации, аналогичные таковым в первой рассматриваемой реализации, не представляются возможными в F# ввиду отсутствия в языке специфических низкоуровневых конструкций. Наивная реализация на OpenCL C участвует в эксперименте с целью сравнения скорости работы OpenCL-кода и аналогичного кода, транслированного из F#.

Запуск производился средствами Brahma.FSharp на видеокарте NVIDIA GeForce GT 755M с тактовой частотой графического процессора 980 МГц и памятью 2048 МБ. Результаты экспериментов представлены на рисунке 3.

Как видно на диаграмме, наивные реализации на F# и OpenCL C почти не отличаются по производительности, в то время как оптимизированная реализация показывает почти трёхкратный прирост производительности на больших матрицах. Это подтверждает целесообразность переиспользования OpenCL-кода, оптимизированного с помощью низкоуровневых конструкций, в высокоуровневом программировании видеопроцессоров.

Заключение

В ходе работы получены следующие результаты:

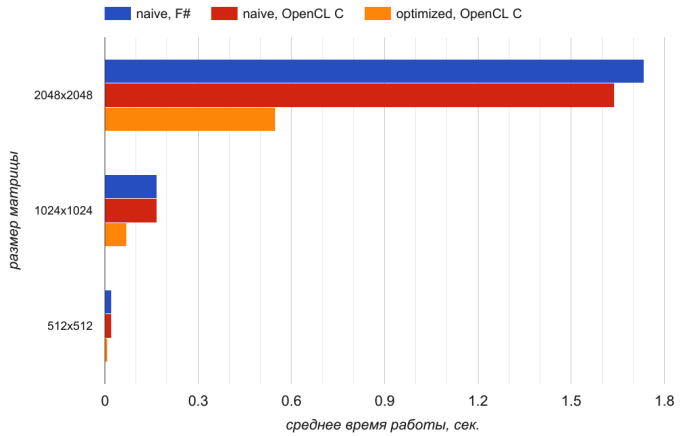


Рис. 3: Результаты экспериментов

– исследован механизм провайдеров типов F#;
– реализован лексический и синтаксический анализатор заголовкой OpenCL-функций;
– реализован механизм типизированного вызова OpenCL-функций в коде на F# на основе провайдера типов;
– проведено экспериментальные исследование работы реализованного модуля.

Код реализованного модуля, включающего лексический, синтаксический анализатор и провайдер типов, находится на сайте `https://github.com/YaccConstructor/Brahma.FSharp`. В указанном репозитории автор принимал участие под учётной записью ksmirenko.

В дальнейшем планируется исследовать возможность применения реализованного механизма при решении задач синтаксического анализа графов с использованием современного алгоритма перемножения разреженных матриц [15]. Соответствующий проект разрабатывается на кафедре системно-

го программирования математико-механического факультета СПбГУ.

Список литературы

1. ANTLR. Another Tool for Language Recognition. — 2014. — URL: `http://http://www.antlr.org/` (online; accessed: 14.05.2017).
2. Brahma.FSharp. Brahma.FSharp // Brahma.FSharp official page. — URL: `http://yaccconstructor.github.io/Brahma.FSharp/` (online; accessed: 14.05.2017).
3. CUSP. CUSP // CUSP official page. — URL: `https://cusplibrary.github.io/` (online; accessed: 14.05.2017).
4. Cocco Gabriele. FSCL: Homogeneous programming and execution on heterogeneous platforms : Ph. D. thesis / Gabriele Cocco ; University of Pisa. — 2014.
5. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming / Peng Du, Rick Weber, Piotr Luszczek et al. // Parallel Comput. — 2012. — Vol. 38, no. 8. — P. 391–407.
6. FsLexYacc. FsLex. — URL: `http://fsprojects.github.io/FsLexYacc/fslex.html` (online; accessed: 14.05.2017).
7. Fung J., Tang F., Mann S. Mediated reality using computer graphics hardware for computer vision // Proceedings. Sixth International Symposium on Wearable Computers,. — 2002. — P. 83–89.
8. Group Khronos. OpenCL // The open standard for parallel programming of heterogeneous systems. — URL: `http://www.khronos.org/opencl/` (online; accessed: 14.05.2017).
9. Group Khronos. The OpenCL C Specification. — 2016. — URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf` (online; accessed: 14.05.2017).
10. Johnson S. C. Yacc: Yet Another Compiler Compiler // Computing Science Technical Report. — 1975.
11. ManagedCUDA. ManagedCUDA // ManagedCUDA official page. — URL: `https://kunzmi.github.io/managedCuda/` (online; accessed: 14.05.2017).
12. MyGEMM. Code appendix to an OpenCL matrix-multiplication tutorial. — URL: `https://github.com/CNugteren/myGEMM` (online; accessed: 14.05.2017).

13. Network Microsoft Developer. Type Providers. — 2016. — URL: https://docs.microsoft.com/en-us/dotnet/articles/fsharp/tutorials/type-providers/ (online; accessed: 14.05.2017).
14. Nvidia. CUDA // Parallel Programming and Computing Platform. — URL: http://www.nvidia.com/object/cuda_home_new.html (online; accessed: 14.05.2017).
15. Polok Lukas, Ila Viorela, Smrz Pavel. Fast Sparse Matrix Multiplication on GPU // Proceedings of the Symposium on High Performance Computing. — HPC '15. — Society for Computer Simulation International, 2015. — P. 33–40. — URL: http://dl.acm.org/citation.cfm?id=2872599.2872604.
16. Programming languages – C : Standard / International Organization for Standardization, International Electrotechnical Commission : 1999.
17. QuantAlea. Alea GPU // QuantAlea official page. — URL: http://www.quantalea.com/ (online; accessed: 14.05.2017).
18. Tihon Sergey. F# Type Providers Development Tips. — 2016. — URL: https://sergeytihon.com/2016/07/11/f-type-providers-development-tips-not-tricks/ (online; accessed: 14.05.2017).
19. Wirth Niklaus. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? // Commun. ACM. — 1977. — Vol. 20, no. 11. — P. 822–823.
20. YaccConstructor. YaccConstructor // YaccConstructor official page. — URL: http://yaccconstructor.github.io (online; accessed: 14.05.2017).
21. YaccConstructor. YARD // YaccConstructor official page. — 2015. — URL: http://yaccconstructor.github.io/YaccConstructor/yard.html (online; accessed: 14.05.2017).
22. Кириленко ЯА, Григорьев СВ, Авдюхин ДА. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости СПбГПУ: информатика, телекоммуникации, управление. — 2013. — no. 174. — P. 94–98.