Правительство Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Полубелова Марина Игоревна

Использование сертифицированных библиотек в несертифицированных системах

Курсовая работа

Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
магистр информационных технологий, ст. преп. Григорьев С. В.

Санкт-Петербург
2016

Оглавление

Введение

Сертификационное программирование [2] позволяет доказывать, что программа соответствует своему формальному описанию. Проверка корректности программы происходит статически, что позволяет гарантировать, что программа всегда работает так, как написано в спецификации. Следовательно, нет необходимости в тестировании сертифицированных программ. Для данного направления широко используются средства для интерактивного доказательства и автоматической проверки теорем (proof assistants). В качестве примеров proof assistants можно привести Coq [16], Agda [15], F* [3], Idris [19] и Epigram [17]. В данной работе используется инструмент F*, так как он является единственным инструментом [12], который позволяет одновременно писать программы и их доказывать.

Области, в которых применяется сертификационное программирование, в основном связаны с формализацией математических объектов, обеспечением безопасного доступа к данным, доказательством свойств языков программирования. Классическими примерами являются обмен сообщениями с помощью криптографических протоколов и распределенные системы, для которых нужно гарантировать, что данные доступны пользователям согласно их привилегиям, а также не нарушаются свойства самих данных (например, файл доступен только для чтения, а над ним выполняется операция записи). Однако доказательство корректности программы требуется не только в рассмотренных областях, но и при разработки остальных алгоритмов. Использование такого подхода увеличит надежность разрабатываемых систем и уменьшит трудозатраты на тестирование.

Системы, обеспечивающие интерактивное доказательство теорем, позволяют доказывать корректность алгоритма только в рамках своей системы. При извлечении проверифицированного кода в другой язык программирования теряется смысл верификации: либо отсутствуют некоторые необходимые проверки, либо подмножество исходного языка не может быть выражено в терминах целевого языка (например, кванторы существования и всеобщности). Даже если написать программу полностью на языке, который использует верификатор, то на практике часто требуется взаимодействие с кодом, который никак не проверифицирован.

Рассмотрим пример, демонстрирующий одну из указанных проблем. В листинге 1 доказана корректность функции `f` с помощью инструмента F* [3]. Ключевое слово `val` используется для описания сигнатуры функции `f`; ключевое слово `Tot` означает, что вычисленный результат всегда имеет тип int без каких-либо эффектов вида: вхождение в бесконечный цикл, возникновение исключений (exceptions), взаимодействие с вводом или выводом. После извлечения функции `f`, написанной на F*, в код на F# [21], представленный в листинге 2, потеряны некоторые свойства функции `f`, а именно, что она имеет эффект `Tot`. Теперь компилятор языка F# позволит вызвать

эту функцию с некорректными, с точки зрения F*, параметрами, например, указанными в листинге 3. Хотя с точки зрения языка F# все будет правильно: для функции g компилятор выведет тип `int -> int`, несмотря на то, что в ветке с `else` происходит вызов исключения.

```
val f : g : (int -> Tot int) -> int -> Tot int
let f g x = g x
```

Листинг 1: Пример проверифицированной функции в F*

```
let f : (int -> int) -> int -> int =
          (fun (g : int -> int) (x : int) -> g x)
```

Листинг 2: Код на F#, полученный из листинга 1

```
    f (fun x: int -> if x > 0
                     then x + 1
                     else failwith "Expected non negative number!" ) 5
```

Листинг 3: Пример вызова функции f

В данной работе рассматривается задача извлечения проверифицированного кода в целевой язык программирования с сохранением его надежности. Для этого необходимо выделить подмножество языка F*, которое можно выразить с помощью конструкций языка F#. При этом генерация всех динамических проверок, например, на соответствие структуре данных, значительно снизит производительность системы, поэтому необходимо решение, которое будет удовлетворять не только требованиям надежности, но и производительности.

1. Постановка задачи

Целью данной работы является обеспечение надежного использования сертифицированных библиотек в несертифицированных системах. Для её достижения были поставлены следующие задачи:

- проверифицировать структуру данных и операции над ней;
- из кода, проверифицированного с помощью инструмента F*, получить код, написанный на языке программирования F#;
- проанализировать получившийся код;
- оценить применимость инструмента F* для задач практического характера на примере верификации преобразования грамматики в нормальную форму Хомского;
- предложить решение обнаруженных проблем.

2. Обзор

2.1. Инструмент F*

В данной работе для верификации программ выбран функциональный язык программирования F* [3, 12], который используется как proof assistant и как язык общего назначения, ориентированный на верификацию. Его система типов включает в себя: полиморфизм, зависимые типы, monadic effects, refinement types и вычисление слабейших предусловий [11]. Указанные свойства позволяют написать спецификацию программы точно и компактно [18]. Для доказательства того, что программа соответствует спецификации, система проверки типов инструмента F* использует SMT solving (Z3 [20]) и свойства, написанные пользователем. Проверифицированный код можно извлечь в языки программирования F# и OCaml.

Кроме того, что язык F* является единственным инструментом [12], который позволяет одновременно писать программы и их доказывать, данный инструмент обладает еще рядом отличительных свойств.

- Поддержка примитивных эффектов, таких как состояние, исключения, расходящиеся функции и ввод/вывод:
  - `Tot t` — эффект вычислений, при котором гарантируется, что вычисленный результат имеет тип `t` без каких-либо эффектов вида: вхождение в бесконечный цикл, возникновение исключений, взаимодействие с вводом или выводом;
  - `ML t` — эффект вычислений, при котором вычисления могут иметь произвольный эффект (это может быть, например, циклом, изменение состояния кучи, вызов исключения), но если результат вычислений получен, то он всегда имеет тип `t`;
  - `Dv t` — эффект вычислений, при котором вычисления могут расходиться;
  - `ST t` — эффект вычислений, при котором вычисления могут расходиться, происходить операции чтения и записи или выделения новых ссылок в куче;
  - `Exn t` — эффект вычислений, при котором вычисления могут расходиться или происходить вызовы исключений.

  Эффекты `{Tot, ML, Dv, ST, Exn}` образуют решетку, в которой элемент `Tot` расположен внизу, элемент `ML` — наверху, а элемент `ST` никак не связан с элементом `Exn`.

- Способ доказательства завершаемости рекурсивных функций: значения аргументов функций должны убывать в лексикографическом порядке. Последнее

можно переопределить с помощью ключевого слова `decreases` и указав параметры, которые будут соответствовать завершаемости функции.

- Написание спецификации программы с помощью зависимых и уточняющих типов.
  - Зависимые типы имеют следующую структуру:
    $x_1 : t_1 \to \ldots \to x_n : t_n[x_1 \ldots x_{n-1}] \to E\ t[x_1 \ldots x_n]$.
    Нотация $t[x_1 \ldots x_n]$ обозначает, что переменные $x_1 \ldots x_n$ могут свободно входить в аргумент t. E содержит эффект вычисления тела функции.
  - Уточняющие (refinement) типы имеют следующий вид:
    $x : t\{\phi(x)\}$ — добавляется ограничение к типу t, а именно, элементы x должны удовлетворять предикату $\phi(x)$.

В инструменте F* существует два подхода к доказательству свойств программы: либо с помощью обогащения типов функции (intrinsic style), либо отдельно формулировать лемму (extrinsic style). Рассмотрим использование этих двух подходов на примере конкатенации двух списков, с доказательством того, что длина результирующего списка равна сумме длин исходных списков:

```
val append: l1:list 'a -> l2:list 'a
            -> Tot (l:list 'a{length l=length l1+length l2})

let rec append l1 l2 =
    match l1 with
    | [] -> l2
    | hd :: tl -> hd :: append tl l2
```

Листинг 4: Верификация функции append, использующая intrinsic style

```
val append_len: l1:list 'a -> l2:list 'a
    -> Lemma (requires True)
             (ensures (length (append l1 l2) = length l1 + length l2)))

let rec append_len l1 l2 =
    match l1 with
    | [] -> ()
    | hd::tl -> append_len tl l2
```

Листинг 5: Верификация функции append, использующая extrinsic style

Ключевые слова `requires` и `ensures` соответствуют предусловию и постусловию леммы. Последний подход используется также для взаимодействия F* с SMT solver.

Например, если в листинг 5 добавить `[SMTPat (append l1 l2)]`, то при каждом использовании `append l1 l2` и при условии выполнения предусловий леммы, будут доступны свойства, доказанные в постусловии леммы.

Таким образом, F* позволяет писать программы, их спецификацию с использованием уточняющих и зависимых типов, а также верифицировать их с помощью SMT solver или интерактивных доказательств. Программы, написанные на F*, могут быть извлечены в язык программирования F# или OCaml. Однако существующий механизм извлечения, схожий с [9], не поддерживает зависимые типы, полиморфизм высших порядков и ghost-вычисления.

2.2. Верификация преобразования грамматики в НФХ

Введем несколько понятий, которые дальше будут использоваться.

Определение 2.1 ***Грамматика*** *— способ описания формального языка, представляющего собой четверку* $\Gamma = \langle \Sigma, N, S \in N, P \subset N^+ \times (\Sigma \cup N)^* \rangle$*, где*

- Σ *— алфавит, элементы которого называют терминалами,*
- N *— множество, элементы которого называют нетерминалами,*
- S *— стартовый символ грамматики,*
- P *— набор правил вывода* $\alpha \to \beta$*.*

Определение 2.2 ***Контекстно-свободная грамматика*** *— это грамматика, в которой правила имеют вид* $A \to \alpha$*, где* A *— нетерминальный символ,* α *— последовательность терминальных и нетерминальных символов (*α *также может быть пустой строчкой).*

Определение 2.3 *Грамматикой в* ***нормальной форме Хомского*** *называется контекстно-свободная грамматика, в которой могут содержаться правила только следующего вида:*

- $A \to BC$
- $A \to a$
- $S \to \varepsilon$,

где A, B, C *— нетерминальные символы,* a *— терминальный символ,* S *— стартовый символ,* ε *— пустая строка. При этом ни* B, C *не являются стартовыми символами.*

Теорема 1 *Любую контекстно-свободную грамматику G можно привести к нормальной форме Хомского.*

Для приведения контекстно-свободной грамматики в нормальную форму Хомского необходимо выполнить следующие шаги:

- Удаление длинных правил. Т.е. замена всех правил вида $A \to X_1 X_2 \dots X_k$, где $k \geq 3$, на правила $A \to X_1 A_1$, $A_1 \to X_2 A_2$, …, $A_{k-2} \to X_{k-1} X_k$.
- Удаление ε-правил. Т.е. правил вида $A \to \varepsilon$.
- Удаление цепных правил. Т.е. правил вида $A \to B$, где A, B — нетерминалы.
- Переименование терминалов в нетерминалы в неподходящих правилах вида $A \to bC$, $A \to Bc$, $A \to BC$.

Стоит отметить, что порядок выполнения преобразований важен. Первое правило должно быть выполнено перед вторым, иначе время нормализации ухудшится до $O(2^{|G|})$. Третье правило идет после второго, потому что после удаления ε–правил могут появиться новые цепные правила. При таком порядке выполнения шагов размеры грамматики возрастают полиномиально. При этом результирующая грамматика G' и исходная грамматика G порождают эквивалентный язык, т.е. $L(G') = L(G)$.

Существует верификация преобразования грамматики в нормальную форму Хомского с помощью инструмента Coq [6] и Agda [5]. Однако в первой работе используется представление правил, которое не позволяет напрямую извлекать выполняемый код из проверифицированного кода. То есть они используют декларативный подход вместо алгоритмического, что позволяет формальную теорию напрямую отразить в верификации. Во второй работе реализуется алгоритмический подход и для доказательства эквивалентности результирующей G' и исходной G грамматики используется функция, которая конвертирует дерево разбора грамматики G в дерево разбора грамматики G' и обратно. Такая функция в одну сторону является тождественной, а в другую — идемпотентной. Для построения деревьев разбора используется сертифицированная реализация алгоритма CYK [4], которая не строит весь лес разбора и не понятно, какие деревья разбора были потеряны в ходе анализа. Для решения этой проблемы авторы планирует поддержать именованные правила.

3. Основная часть

3.1. Верификация библиотеки dList

Структура данных dList [7] — это вариант представления структуры данных List, благодаря которому конкатенация списков возможна за константное время.

Описание структуры dList на языке F* выглядит следующим образом:

```
type dList 't =
    | Nil : dList 't
    | Unit : 't -> dList 't
    | Join : dList 't -> dList 't  -> nat -> dList 't
```

В dList используется вспомогательный параметр типа `nat` (алиас для неотрицательных целых чисел) для хранения длины списка.

В данной работе для рассматриваемой структуры данных проверифицированы следующие операции:

- *head* — возвращает первый элемент списка;
- *tail* — возвращает список без первого элемента;
- *length* — возвращает длину списка;
- *append* — возвращает результат конкатенации двух списков;
- *cons* — прибавляет элемент к началу списка;
- *snoc* — прибавляет элемент к концу списка;
- *fold* — свертка списка.

Уже доказательство корректности первой операции требует изменения в определении структуры данных dList: необходимо проверять, является ли пустым один из аргументов `Join`:

```
val isCorrectJoined : l : dList1 't -> Tot bool
let rec isCorrectJoined l =
        match l with
        | Nil -> true
        | Unit x -> true
        | Join Nil _ _ -> false
        | Join x y l -> isCorrectJoined x && isCorrectJoined y
```

```
type dList1 't = l : dList 't {isCorrectJoined l}
```

В дальнейшем используется последнее определение для структуры dList. Доказательства корректности операций представлены ниже.

- Для операции *head* в случае, когда аргумент является пустым списком `Nil`, результат не определен. Для доказательства этот факт можно интерпретировать по-разному, здесь же считается, что операция *head* всегда применяется к листу, состоящего хотя бы из одного элемента.

```
val isCons: dList 'a -> Tot bool
let isCons xs =
    match xs with
    | Unit x -> true
    | Join x y z -> true
    | _ -> false

val head : l : dList 't {isCons l} -> Tot 't
let rec head l =
    match l with
    | Unit x -> x
    | Join x y _ -> head x
```

- Для операции *append* обычно проверяют, что длина результирующего списка равна сумме длин исходных списков.
- Для операции *length* обычно проверяют, что результат является неотрицательным числом.
- Для операции *tail* можно доказать уже некоторые свойства. Например, конкатенация результата операции *head* с результатом операции *tail* есть исходный список, то есть `append (head l) (tail l) = l`.

```
val tail : l:dList 't -> Tot(l1:dList 't
{ (length l = 0 ==> length l1 = 0) \/
  (length l > 0 ==> length l1 = length l - 1 /\ append (Unit (head l)) l1 = l)})
let tail l = if length l = 0 then Nil else step l Nil
```

```
val step: xs:dList 't -> acc:dList 't -> Tot(l: dList 't)
let rec step xs acc =
    match xs with
    | Nil -> acc
    | Unit _ -> acc
    | Join x y _ ->
        step x (match y with
                | Nil -> acc
                | _ -> match acc with
                       | Nil -> y
                       | _ -> Join y acc (length y + length acc))
```

- Для операции *cons* тоже доказываются некоторые свойства:
  - результат содержит хотя бы один элемент;
  - первым элементом результирующего списка является добавленный элемент;
  - хвостом результирующего списка является исходный список;
  - полученный результат можно получить и другим способом: с применением операции *append*;
  - длина списка увеличилась на 1.

```
val cons: hd : 't -> l: dList 't -> Tot(l1 : dList 't
{isCons l1 /\ head l1 = hd /\ tail l1 = l /\
l1 = append (Unit hd) l /\ length l + 1 = length l1})
let cons hd l =
    match l with
    | Nil -> Unit hd
    | _ -> Join (Unit hd)  l (length l + 1)
```

- Доказанные свойства операции *snoc* совпадают с некоторыми свойствами операции *cons*:

```
val snoc: tl : 't -> l: dList 't -> Tot(l1 : dList 't
{isCons l1 /\ l1 = append l (Unit tl) /\ length l + 1 = length l1})
let snoc tl l =
    match l with
    | Nil -> Unit tl
    | _ -> Join l (Unit tl) (length l + 1)
```

- Для операции *fold* используется реализация[1], которая имеет линейную сложность. Одной из особенностью инструмента F* является способ доказательства завершаемости рекурсивной функции, а именно, проверяется, что значения аргументов функции уменьшаются в лексикографическом порядке. Однако можно явно указывать, какой параметр соответствует завершаемости алгоритма. Здесь таким параметром является количество "связующих" элементов списка (`Nil`, `Unit`, `Join`). Функция `cntdList` возвращает указанный параметр для списка типа `dList`, а функция `cntList` — для обычного списка, каждый элемент которого является список типа `dList`.

```
val fold : ('a -> 't -> Tot 'a) -> 'a -> dList 't -> Tot 'a
let fold f state l = walk [] l state f

val finish : rights : list (dList1 't) -> xs : 'a -> f : ('a -> 't -> Tot 'a)
-> Tot 'a  (decreases %[cntList rights; 1])
val walk : rights : list (dList1 't)  -> l : dList1 't -> xs : 'a
-> f : ('a -> 't -> Tot 'a)  -> Tot 'a (decreases %[cntdList l + cntList rights; 0])
let rec walk rights l xs f =
    match l with
    | Nil         -> finish rights xs f
    | Unit x      -> finish rights (f xs x) f
    | Join x y _  -> walk (y::rights) x xs f
and finish rights xs f =
    match rights with
    | []     -> xs
    | hd::tl -> walk tl hd xs f
```

3.2. Кодогенерация из F* в F#

Инструмент F* позволяет извлекать код в языки программирования F# и OCaml. В данном разделе рассматриваются проблемы, которые возникают в целевом коде, на примере проверифицируемой структуры данных dList.

В предыдущем разделе были описаны два типа для структуры данных dList: один без ограничений, другой с ними. В целевом коде эта информация не сохранилась:

[1]Реализация взята с сайта http://stackoverflow.com/questions/5324623/functional-o1-append-and-on-iteration-from-first-element-list-data-structure/5334068#5334068

```
type 't dList =
| Nil
| Unit of 't
| Join of 't dList1 * 't dList1 * Prims.nat

type 't dList1 = 't dList
```

Аналогично и с функциями: ограничения на уровне типов никак не отразились в целевой код. Например, корректность функции head была доказана в предположении, что она применяется только к спискам, состоящим хотя бы из одного элемента. Компилятор языка F# укажет только на не полный pattern matching.

```
let rec head = (fun ( l  :  't dList ) -> (match (l) with
| Unit (x) -> begin
x
end
| Join (x, y, _5_83) -> begin
(head x)
end))
```

Одним из возможных решений указанных проблем является генерация всех динамических проверок. С одной стороны, такой подход нельзя осуществить полностью, так как часть языковых конструкций F* не выразима в терминах языка F#. С другой стороны, динамическая проверка на корректность выполнения каждой операции значительно снизит производительность всей программы. Существующие подходы к решению данной проблемы в других proof assistants заключаются либо в расширении синтаксиса целевого языка [1], либо в использовании особенностей исходного языка [14, 13] (например, аксиоматический подход для реализации смешанных вычислений в Coq). В последней работе обсуждается существование изоморфизма между структурами зависимых типов и простых структур, что является более общим подходом по сравнению с работой [10](dependent interoperability) и вероятно станет дальнейшим направлением данной работы.

3.3. Верификация преобразования грамматики в НФХ в F*

Для верификации преобразования грамматики необходимо выполнить следующее:

- доказать тотальность каждого преобразования;
- доказать, что каждое преобразование соответствует своей спецификации (например, после удаления цепных правил в результирующей грамматике их нет);

- доказать эквивалентность исходной и результирующей грамматик.

За основу верификации алгоритма нормализации грамматики взята реализация, выполненная в проекте YaccConstructor[2] [8], который является модульным инструментом для проведения лексического анализа и синтаксического разбора, а также платформой для исследования и разработки генераторов лексических и синтаксических анализаторов. Основной язык разработки — F#, что как раз позволит оценить сложность переносимости кода из программы, написанной на F#, в инструмент F*.

В проекте YaccConstructor правила грамматики представляются структурой, которая имеет поля для имени и правой части продукции, аргументов и мета-аргументов, а также является ли это правило стартовым. Правой частью продукции является списком терминальных и нетерминальных символов (PSeq, PRef, PAlt, PSome, PMany и т.д.). Сама грамматика представляется списком правил. Функция нормализации грамматики выглядит следующим образом:

```
let toCNF (ruleList: list (Rule _ _)) =
    ruleList
    |> splitLongRule
    |> deleteEpsRule
    |> deleteChainRule
    |> renameTerm
```

Так как язык программирования F# является мультипарагдимальным, то есть поддерживает не чисто функциональный стиль программирования, при котором возможно использование переменных (mutable variables), то для доказательства тотальности каждого преобразования необходимо переписать функции без их использования.

Рассмотрим доказательство тотальности для каждого преобразования, тогда тотальность функции `toCNF` получится как результат композиции тотальных функций. При этом используется библиотека `List.Tot` из F*, в которой доказаны тотальность основных операций над списками.

Удаление длинных правил (splitLongRule)

Функция `splitLongRule` принимает на вход список правил и к каждому правилу применяет функцию `cutRule`, которая возвращает список коротких правил:

```
val splitLongRule: list (Rule 'a 'b) -> Tot (list (Rule 'a 'b))
let splitLongRule ruleList = List.Tot.collect (fun rule -> cutRule rule []) ruleList
```

[2]Репозиторий проекта YaccConstructor `https://github.com/YaccConstructor`

Для того чтобы доказать, что функция `splitLongRule` возвращает список коротких правил, необходимо отдельно доказать, что конкатенация списков коротких правил есть список коротких правил.

Рекурсивная функция `cutRule` принимает на вход правило и если его длина больше двух, то происходит взятие первых двух элементов от инвертированного списка (`[List.Tot.hd revEls; List.Tot.hd (List.Tot.tl revEls)]`), из которых создается новое правило. Созданное правило добавляется к списку `List.rev (List.Tot.tl (List.Tot.tl revEls))`, с которым вновь вызывается функция `cutRule`. Сигнатура функции `cutRule` имеет следующий вид:

```
val cutRule: rule : (Rule 'a 'b)
 -> acc:(list (Rule 'a  'b))
          {List.Tot.for_all (fun x -> lengthBodyRule x <= 2) acc}
 -> Tot (res:(list (Rule 'a 'b))
               {List.Tot.for_all (fun x -> lengthBodyRule x <= 2) res})
  (decreases %[lengthBodyRule rule])
```

Условием завершаемости функции `cutRule` является уменьшение длины обрабатываемого правила. Переменная `acc` накапливает результирующий список коротких правил, который возвращается, когда длина входного правила меньше двух.

Одним из преимуществ инструмента F* является то, что когда в условии `if` (которое можно перенести в качестве ограничения на тип принимаемых аргументов функцией) формулируется проверка на длину списка, например, больше двух, то это позволяет взять первые два элемента списка. В то время как использование функции `nth`, отвечающей за взятие n-го элемента списка, затрудняет процесс доказательства из-за того, что возвращает тип `Option`:

```
val nth: list 'a -> nat -> Tot (option 'a)
let rec nth l n = match l with
  | []     -> None
  | hd::tl -> if n = 0 then Some hd else nth tl (n - 1)
```

Для доказательства тотальности функции `cutRule` потребовалась также доказать вспомогательные леммы: длина инвертированного списка совпадает с длиной исходного списка; длина хвоста списка равна длине исходного списка, уменьшенного на 1; конкатенация списков коротких правил есть список коротких правил.

Удаление ε-правил (deleteEpsRule)

Функция `deleteEpsRule` принимает на вход список правил и возвращает список правил, в котором нет ε-правил и правил вида $A \to A$.

```
val deleteEpsRule: list (Rule 'a  'b) -> Tot (list (Rule 'a  'b))
let deleteEpsRule ruleList =
    let rulesFilter =
        List.Tot.filter
            (fun r -> not (match r.body with PSeq([],_,_) -> true | _ -> false))
            (newRules ruleList) in
    deleteTrashRule rulesFilter
```

Функция `deleteTrashRule` удаляет правила вида $A \to A$. Функция `newRules` принимает на вход список правил и если правило является ε-правилом, то вызывается функция `newRule`, которая возвращает список всех возможных вариантов правил, в которых либо присутствует, либо удален каждый из нетерминалов, который является ε-порождающим.

Для получения всех возможных вариантов правил, в правой части которых либо присутствует, либо отсутствует какой-то нетерминал, необходима генерация всех подмножеств множества `[1..n]`, где `n` — длина правой части продукции. В F* отсутствует range-генератор, поэтому вместо записи `[1..n]` пришлось писать функцию, которая имеет следующий вид:

```
val listfromto_acc: a:int {a >= 1} -> b:int {b >= a} -> list int -> Tot (list int)
let rec listfromto_acc a b acc =
        if (a = b) then List.Tot.append [a] acc
        else listfromto_acc a (b - 1) (List.Tot.append [b] acc)

val listfromto: a:int {a >= 1} -> b:int {b >= a} -> Tot (list int)
let listfromto a b = listfromto_acc a b []
```

Удаление цепных правил (deleteChainRule)

Функция `deleteChainRule` принимает на вход список правил и если правая часть правила состоит из одного нетерминала, то вызывается функция `newRule`, которая из нескольких цепных правил вида $A_1 \to A_2 \to \ldots \to A_n \to a$ создает одно новое правило $A_1 \to a$. В результате получается список правил, в котором отсутствуют цепные правила.

```
val deleteChainRule: list (Rule 'a 'b) -> Tot (list (Rule 'a 'b))
let deleteChainRule ruleList =
    let lengthRuleList = List.length ruleList in
    List.Tot.collect
        (fun rule ->
            match rule.body with
            | PSeq(els, _, _) ->
                (match isNonTerminal els with
                    | true -> newRule ruleList rule lengthRuleList (nameRule rule)
                    | _ -> [rule])
            | _ -> [rule]) ruleList
```

Рекурсивная функция `newRule` принимает на вход список всех правил; правило, которое является цепным и имя нетерминала, находящегося в правой части продукции передаваемого правила; длину результирующего списка правил. Последний параметр является условием завершаемости функции `newRule`: при каждом новом рекурсивном вызове количество правил уменьшается на 1, так как удаляется одно цепное правило. Функция `newRule` имеет следующий вид:

```
val newRule: list (Rule 'a 'b) -> Rule 'a 'b -> lengthRL : nat -> string
             -> Tot (list (Rule 'a 'b)) (decreases %[lengthRL])
let rec newRule ruleList mainRule lengthRL name =
    List.Tot.collect
    (fun rule ->
        if (rule.name.text = name) && (lengthRL > 0) then
            if isNonTerminal rule then
                match rule.body with
                | PSeq(els, _, _) ->
                        newRule ruleList mainRule (lengthRL - 1) (nameRule rule)
                | _ -> newRule ruleList mainRule (lengthRL - 1) ""
            else
                [{mainRule with body = bodyChange mainRule rule}]
        else []) ruleList
```

Функция `bodyChange` меняет правую часть правила `mainRule` на правую часть правила `rule`.

Переименование терминалов в нетерминалы в неподходящих правилах (renameTerm)

Функция `renameTerm` принимает на вход список правил и если правило не находится в нормальной форме Хомского (НФХ), то вызывается функция `renameRule`. В результате возвращается список правил, которые находятся в НФХ.

```
val renameTerm: list (Rule 'a 'b) -> Tot (list (Rule 'a 'b))
let renameTerm ruleList =
    renameTerm_acc ruleList []
            (fun rule acc -> if isCNF rule then [rule] else renameRule rule acc)
```

Функция `isCNF` проверяет, находится ли правило в НФХ и имеет следующий вид:

```
val isCNF: Rule 'a 'b -> Tot bool
let isCNF rule =
    match rule.body with
    | PSeq(els,_,_) -> (List.Tot.length els = 1) ||
                       (List.Tot.length els = 2 && isNonTerminals els) ||
                       (List.Tot.length elements = 0 && rule.isStart)
    | _ -> false
```

Функция `renameTerm_acc` используется вместо функции `List.Tot.collect` для того, чтобы можно было накапливать текущий результирующий список правил, который использует функция `renameRule` для проверки уникальности имен для новых правил:

```
val renameTerm_acc: list (Rule 'a 'b) -> list (Rule 'a 'b) ->
    (Rule 'a 'b -> list (Rule 'a 'b) -> Tot (list (Rule 'a 'b)))
    -> Tot (list (Rule 'a 'b))
let rec renameTerm_acc ruleList acc f =
        match ruleList with
        | [] -> acc
        | hd::tl -> renameTerm_acc tl (List.Tot.append acc (f hd acc)) f
```

Функция `renameRule` на вход принимает правило, длина которого равна 2, и текущий результирующий список правил. Если исходное правило является неподходящим, то есть хотя бы один элемент из правой части правила является терминалом, то происходит создание нового правила. Например, если правило имело вид $A \to bC$, то функция `renameRule` вернет два правила: $A \to BC$ и $B \to b$. Так как функция `renameRule`

применяется только для правил длины 2, то можно использовать List.Tot.hd два раза, чтобы получить оба элемента списка вместо использования функции `nth`, которая возвращает тип `Option`. Сигнатура функции `renameRule` имеет следующий вид:

```
val renameRule: rule : (Rule 'a 'b){lengthBodyRule rule = 2}
                -> list (Rule 'a 'b) -> Tot (list (Rule 'a 'b))
```

Таким образом, в данной работе доказана тотальность каждого преобразования, что позволяет инструменту F* заключить, что функция `toCNF` является тотальной (как композиция тотальных функций), то есть:

```
val toCNF: list (Rule 'a 'b) -> Tot (list (Rule 'a 'b))
let toCNF ruleList =
    let resSplitLongRule = splitLongRule ruleList in
        let resDeleteEpsRule = deleteEpsRule resSplitLongRule in
            let resDeleteChainRule = deleteChainRule resDeleteEpsRule in
                renameTerm resDeleteChainRule
```

При этом в исходном коде использовался оператор pipeline (|>), однако в инструменте F* указанный оператор не сохраняет свойство тотальности [3].

[3] https://github.com/FStarLang/FStar/issues/530

Заключение

При выполнении данной работы были получены следующие результаты:

- описана на F* и проверифицирована структура данных dList и операции над ней;
- проанализирован код на F#, извлеченный из соответствующего кода на F*;
- выявлены проблемы с надежностью кода, извлеченного из проверифицированного кода, из-за отсутствия динамических проверок на входные данные, в предположении которых выполнялись доказательства корректности программы;
- рассмотрены существующие подходы к решению выявленных проблем;
- доказана тотальность преобразования грамматики в нормальную форму Хомского;
- результаты работы вошли в статью "Certified Grammar Transformation to Chomsky Normal Form in F*" (SYRCoSE-2016 Spring/Summer Young Researchers' Colloquium on Software Engineering).

Код верификации преобразования грамматики в нормальную форму Хомского можно найти на сайте `https://github.com/YaccConstructor/YC_FStar`. В указанном репозитории автор принимал участие под учетной записью polubelova.

Дальнейшее направление

Дальнейшем направлением данной работы является реализация подхода, обеспечивающего надежное использование сертифицированных библиотек в несертифицированных системах. Для этого необходимо выделить подмножество языка F*, которое можно выразить в терминах языка F#. При этом необходимо сообщать пользователю, какая часть кода является надежной, а какая нет.

Другим направлением является верификация алгоритмов теории формальных языков и грамматики. В данной работе доказана тотальность преобразования грамматики в нормальную форму Хомского, в дальнейшем необходимо также доказать, что результирующая грамматика обладает нужными свойствами: в ней отсутствуют длинные правила, ε-правила, цепные правила, а также ситуации, когда в правиле встречаются несколько нетерминалов. Важным остается также доказательство того, что результирующая грамматика эквивалентна исходной грамматике.

Список литературы

[1] Chen Juan, Chugh Ravi, Swamy Nikhil. Type-preserving Compilation of End-to-end Verification of Security Enforcement // SIGPLAN Not. — 2010. — Vol. 45, no. 6. — P. 412–423. — URL: http://doi.acm.org/10.1145/1809028.1806643.

[2] Chlipala Adam. Certified programming with dependent types. — 2016.

[3] F* Tutorial. — URL: https://www.fstar-lang.org/tutorial/.

[4] Firsov Denis, Uustalu Tarmo. Certified CYK parsing of context-free languages // Journal of Logical and Algebraic Methods in Programming. — 2014. — Vol. 83, no. 5. — P. 459–468.

[5] Firsov Denis, Uustalu Tarmo. Certified normalization of context-free grammars // Proceedings of the 2015 Conference on Certified Programs and Proofs / ACM. — 2015. — P. 167–174.

[6] Formalization of context-free language theory / Marcus VM Ramos, Ruy JGB de Queiroz, Nelma Moreira, José Carlos Bacelar Almeida // arXiv preprint arXiv:1510.09092. — 2015.

[7] Hughes R J M. A Novel Representation of Lists and Its Application to the Function "Reverse" // Inf. Process. Lett. — 1986. — Vol. 22, no. 3. — P. 141–144. — URL: http://dx.doi.org/10.1016/0020-0190(86)90059-1.

[8] Kirilenko Iakov, Grigorev Semen, Avdiukhin Dmitriy. Syntax analyzers development in automated reengineering of informational system // St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems. — 2013. — no. 174. — P. 94 – 98.

[9] Letouzey Pierre. Extraction in Coq: An overview. — 2008. — P. 359–369.

[10] Osera Peter-Michael, Sjöberg Vilhelm, Zdancewic Steve. Dependent interoperability // Proceedings of the sixth workshop on Programming languages meets program verification / ACM. — 2012. — P. 3–14.

[11] Pierce Benjamin C. Types and Programming Languages. — Cambridge, MA, USA : MIT Press, 2002.

[12] Swamy Nikhil, Hriţcu Cătălin, Keller Chantal. Dependent Types and Multi-Monadic Effects in F* // 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). — ACM, 2016. — P. 256–270. — URL: https://www.fstar-lang.org/papers/mumon/.

[13] Tanter Eric, Tabareau Nicolas. Lost in Extraction, Recovered.

[14] Tanter Éric, Tabareau Nicolas. Gradual Certified Programming in Coq. — 2015. — P. 26–40. — URL: http://doi.acm.org/10.1145/2816707.2816710.

[15] Сайт проекта Agda. — URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[16] Сайт проекта Coq. — URL: https://coq.inria.fr/.

[17] Сайт проекта Epigram. — URL: https://github.com/mietek/epigram2.

[18] Сайт проекта F*. — URL: https://www.fstar-lang.org/.

[19] Сайт проекта Idris. — URL: http://www.idris-lang.org/.

[20] Сайт проекта Z3. — URL: http://z3.codeplex.com/.

[21] Язык программирования F#. — URL: http://fsharp.org/.