Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование
информационных систем

Системное программирование

Полубелова Марина Игоревна

Компиляция сертифицированных F*-программ в робастные Web-приложения

Магистерская диссертация

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

Рецензент:
ООО “ИнтеллиДжей Лабс”
разработчик ПО Мордвинов Д. А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Polubelova Marina

Compiling Verified F* Programs to Robust Web Applications

Master's Thesis

Scientific supervisor:
Associate Professor Grigorev S. V.

Reviewer:
"IntelliJ Labs Co.Ltd"
Software Developer Mordvinov D. A.

Saint-Petersburg
2017

Оглавление

Введение

В последнее время происходит стремительный рост количества устройств, подключаемых к Интернет. Взаимодействие 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 обеспечении. 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 происхо-

дит 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* [15, 25, 4], 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).

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

Листинг 1: Функция вычисления факториала

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

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

Листинг 2: Функция вычисления факториала с эффектом Tot

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).

```
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)
```

Листинг 3: Результат вычисления факториала является положительным числом

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 вычислительного 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.

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

Листинг 4: Результат трансляции функции $fact\_property$

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 зациклит-

```
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)
```

Листинг 5: Результат вычисления факториала является положительным числом

ся. Данный эффект можно отобразить с использованием ключевого слова Dv (см. листинг 6).

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

Листинг 6: Функция вычисления факториала с эффектом Dv

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 $r2$.

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

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

Листинг 8: Функция вычисления факториала с эффектом ML

Таким образом, эффекты $\{Tot, ML, Dv, ST\}$ образуют решетку, в которой $Tot <$

```
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
```

Листинг 7: Функция вычисления факториала с эффектом ST

$Dv < ML$ 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]:

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

Листинг 9: Пример функции, демонстрирующий особенности хранения переменных

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

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

Листинг 10: Неявный вид функции, представленный в листинге 9

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`.

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.

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

Листинг 11: Окружение, которое используется правилами трансляции

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, 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 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 15 и 16), в то время как синтаксис языка JavaScript содержит в себя как выражения, так и операторы (statements) (см. таблицы 17 и 18). В листинге 12 приведен пример OCaml-программы, в которой условное выражение `if-then-else` используется внутри арифметического выражения.

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

Листинг 12: Конструкция $if - then - else$ внутри арифметического выражения

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.

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

Листинг 13: Результат трансляции кода, представленного в листинге 12

Таким образом, правила трансляции для выражений используют два типа функций трансляции. Первый тип $[\,]_\rho$ используется, когда происходит трансляция $expr\_ml \to expr\_js$ и второй тип $[\![\ ]\!]_\rho\ \_\_$, когда происходит трансляция $expr\_ml \to stmt\_js$. Правила трансляции OCaml-выражений представлены в листингах 20 и 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 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:

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

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 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 от-

транслированной программы. Например, в правиле $[\![\ C\ e_1\ \ldots\ e_n\ ]\!]_\rho$ в зависимости от значения имени конструктора C используется замена, указанная в листинге 14.

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

Листинг 14: Частные случаи трансляции конструктора

3. Правила трансляции выражений для сопоставления с образцом

Правила трансляции выражений для сопоставления с образцом представлены в листинге 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 23, можно избавиться от повторения конструкции s_2. Пример одного из подхода, позволяющий это сделать, приведен в листинге 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 25 , целевого языка — в листинге 26. Правила трансляции типов представлены в листинге 27. В зависимости от значения имени конструктора можно ввести замену, согласованную с листингом 14.

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

Листинг 15: Используемые обозначения

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

Листинг 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 17: Используемые обозначения

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

Листинг 18: Синтаксис используемого в данной работе подмножества языка JavaScript

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

Листинг 19: Правила трансляции типов OCaml-констант в типы JavaScript-констант

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

Листинг 20: Правила трансляции OCaml-выражений в JavaScript-выражения

$$
\begin{array}{lcl}
[\![\ 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\ \Rightarrow\ e\ ]\!]_\rho\ \ f\ s & = & \text{let } f\ =\ ([\ x\ ]_\rho) \Rightarrow \{[\![\ e\ ]\!]_{\rho_1}\ \ \_res\ (return\ \_res)\};\ s \\
[\![\ \text{match } e \text{ with } |\ p_i\ \rightarrow\ e_i\ ]\!]_\rho\ \ x\ s & = & [\![\ e\ ]\!]_\rho\ \ fv\_e\ (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}\ \ \_\ (\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\} \text{ else } \{[\![\ e_2\ ]\!]_\rho\ \ x\ None\});\ s
\end{array}
$$

Листинг 21: Правила трансляции OCaml-выражений в JavaScript-операторы

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

Листинг 22: Функция $translate\_match$

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

Листинг 23: Правила трансляции сопоставления с образцом

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

Листинг 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 25: Синтаксис используемого в данной работе набор типов языка OCaml

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

Листинг 26: Синтаксис используемого в данной работе набор типов языка JavaScript, предоставляемых инструментом Flow

$$
\begin{array}{lcl}
[\![\ int\ ]\!]_\rho & = & number \\
[\![\ bool\ ]\!]_\rho & = & bool \\
[\![\ string\ ]\!]_\rho & = & string \\
[\![\ (t_1 * t_2 * \ldots * t_n)\ ]\!]_\rho & = & [[\![\ t_1\ ]\!]_\rho, [\![\ t_2\ ]\!]_\rho, \ldots, [\![\ t_n\ ]\!]_\rho] \\
[\![\ C\ t_1 \ldots t_n\ ]\!]_\rho & = & C <[\![\ t_1\ ]\!]_\rho, [\![\ t_2\ ]\!]_\rho, \ldots, [\![\ t_n\ ]\!]_\rho> \\
[\![\ t_1 \to t_2\ ]\!]_\rho & = & [\![\ t_1\ ]\!]_\rho \Rightarrow [\![\ t_2\ ]\!]_\rho \\
[\![\text{ type } C\ x_1 \ldots x_n\ = & = & \text{type } C < x_1 \ldots x_n > = \\
\{f_1 : t_1, \ldots, f_n : t_n\}\ ]\!]_\rho & & \{\_tag : \text{``}Record\text{''}, \\
 & & \_f_1 :[\![\ t_1\ ]\!]_\rho, \ldots, \_f_n :[\![\ t_n\ ]\!]_\rho\} \\
[\![\text{ type } C\ x_1 \ldots x_n\ =\ t\ ]\!]_\rho & = & \text{type } C < x_1 \ldots x_n > = [\![\ t\ ]\!]_\rho \\
[\![\text{ type } C\ x_1 \ldots x_n\ = & = & \text{type } C_1 < x_1 \ldots x_n > =\ \{\_tag : \text{``}C_1\text{''}, \_1 :[\![\ t_1\ ]\!]_\rho\} \\
\mid C_1 \text{ of } t_1 & & \ldots \\
\ldots & & \text{type } C_n < x_1 \ldots x_n > =\ \{\_tag : \text{``}C_n\text{''}, \_1 :[\![\ t_n\ ]\!]_\rho\} \\
\mid C_n \text{ of } t_n\ ]\!]_\rho & & \text{type } C < x_1 \ldots x_n > =\ C_1 < x_1 \ldots x_n > | \ldots | C_n < x_1 \ldots x_n > \\
[\![\text{ type } C\ x_1 \ldots x_n\ = & = & \text{type } C < x_1 \ldots x_n > = \\
\mid C_1 \text{ of } t_1\ \to\ \ldots\ \to\ t_n\ ]\!]_\rho & & \{\_tag : \text{``}C_1\text{''}, \_1 :[\![\ t_1\ ]\!]_\rho, \ldots, \_n :[\![\ t_n\ ]\!]_\rho\}
\end{array}
$$

Листинг 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.

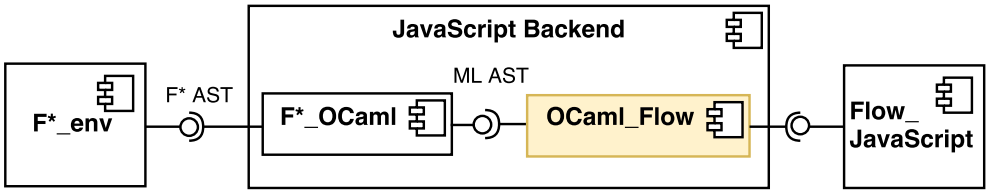

Рис. 1: Диаграмма компонентов реализованного инструмента

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).

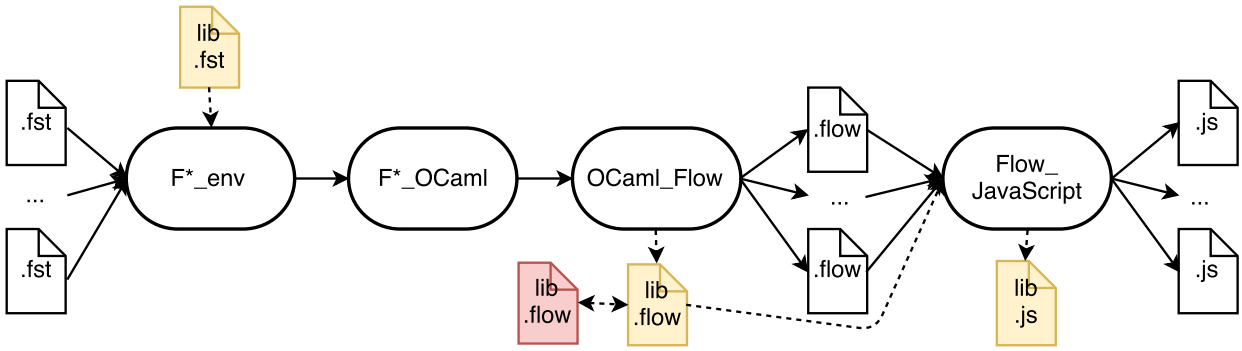

Рис. 2: Процесс построения JavaScript-приложения из F*-программы

4.3. Работа с библиотечными функциями

Программа чаще всего состоит из нескольких модулей и использует библиотечные функции. В F* такие библиотечные 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 $crypto.randomBytes(len)$ 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 миллисекундах на байт, то есть среднее время выполнения функции *encrypt* было разделено на размер сообщения.

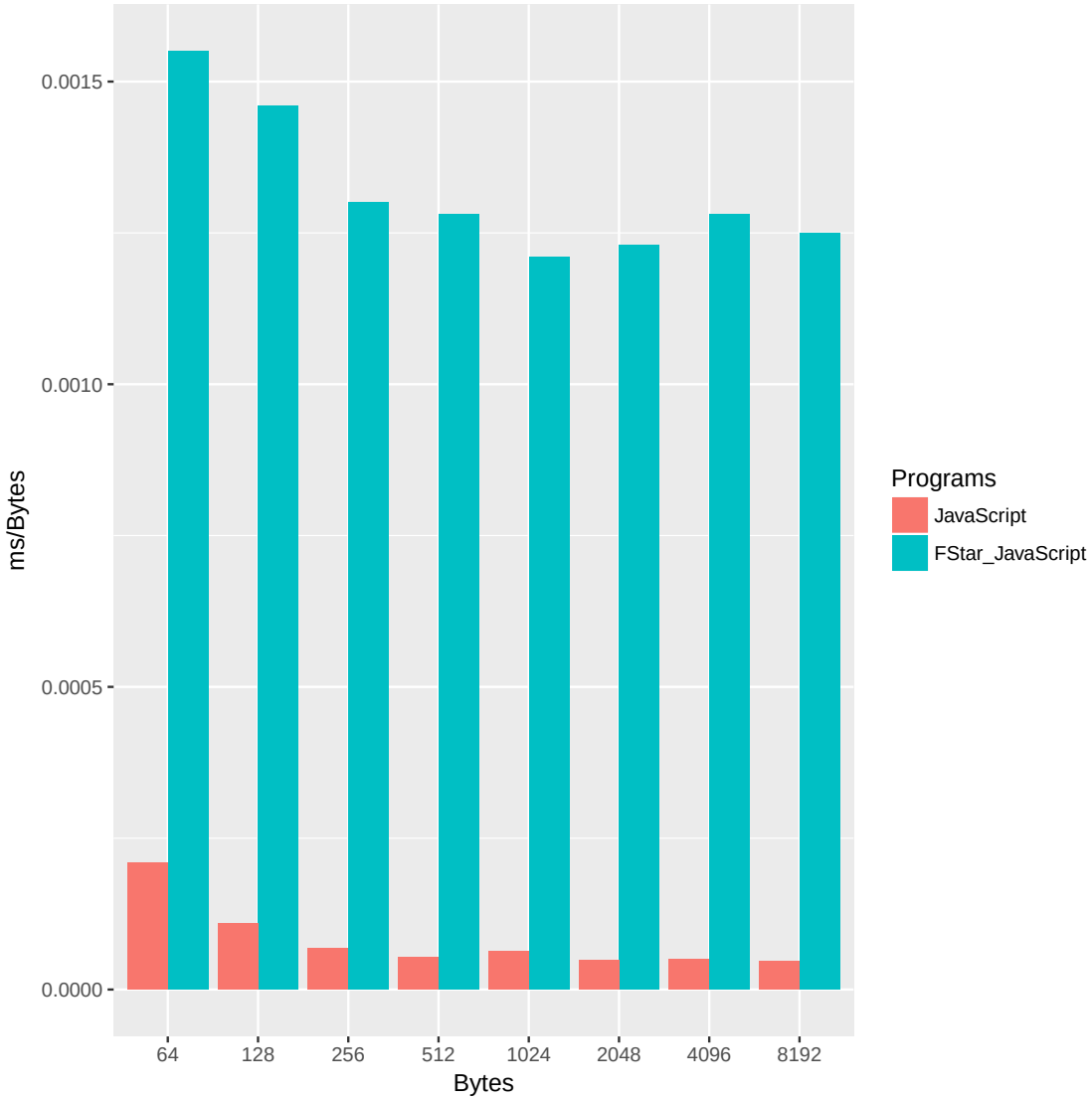

Рис. 3: График сравнения времени выполнения программ JavaScript и F*_JavaScript

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 подмно-

жества 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/.