Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Цириков Семен Алексеевич

Компиляция программ на OCaml в Lua

Отчёт по учебной практике

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

Консультант:
программист "IntelliJ Labs" Косарев Д.С.

Санкт-Петербург
2020

Оглавление

1. Введение **3**

2. Постановка задачи **4**

3. Обзор **5**

3.1. JavaScript 5

3.2. Js_of_ocaml 6

3.3. BuckleScript 6

3.4. Lua 7

4. Реализация **8**

4.1. Архитектура 8

4.2. Стек технологий 8

4.3. Детали реализации 9

4.4. Тестирование 12

5. Заключение **14**

Список литературы **15**

1. Введение

При разработке встроенных систем возможности аппаратного обеспечения становятся существенными ограничениями для разработчика (например, невысокая скорость процессора и малый объём памяти, продиктованные условиями минимизации энергопотребления, физического размера или конечной стоимости продукта) и специфичные требования к ПО: часто применяются сокращённые бинарные сборки программ, в которых отсутствует код, не используемый непосредственно. Эти препятствия осложняют использование многих сторонних библиотек (Qt, boost, glibc и другие). Тем не менее программисты зачастую вынуждены использовать высокоуровневые языки программирования с небольшой RTL — *библиотекой среды исполнения* (порядка нескольких десятков килобайт).

Интерес к функциональному программированию со стороны разработки для встроенных систем обусловлен повышенной надёжностью кода и доступностью применения сложных методов автоматической оптимизации. Однако использование напрямую языков, полноценно реализующих функциональную парадигму, затруднено размерами RTL — необходимо более 200 КБ для OCaml.

Для преодоления разрыва между мощностью исходного языка и ограничениями, накладываемыми целевой архитектурой, можно использовать методы трансляции. С учётом особенностей встроенных систем, предпочтительна именно компиляция, так как отсутствие на устройстве ресурсов для интерпретации или JIT-компиляции может оказаться критическим.

В ходе работы планируется создать инструмент компиляции из OCaml [5] в Lua [2], предварительно изучив существующие аналоги проекта, построив трансляцию основных структур данных OCaml в структуры данных Lua, а также протестировать инструмент на корректность получаемых программ.

2. Постановка задачи

Целью данной работы является разработка инструмента компиляции программ, написанных на языке OCaml, в код на языке Lua.

Для достижения указанной цели были поставлены следующие задачи:

- Провести сравнительный анализ существующих решений:
  - BuckleScript
  - Js_of_ocaml
- Спроектировать трансляцию структур данных OCaml в структуры данных Lua
- Реализовать прототип инструмента, предоставляющий возможность использования следующих конструкций:
  - Объявление и использование переменных
  - Стандартные арифметические, битовые, булевы и строковые операции, а также операции сравнения
  - Условное ветвление, в том числе сопоставление с образцом
  - Цикл
  - Функции и рекурсия, каррирование
  - Обработка исключений
  - Вывод
  - Импорт модулей
- Провести тесты и проверить корректность трансляции

3. Обзор

В данной секции сперва приводятся причины, по которым разработчики могут быть заинтересованы в использовании инструментов кодогенерации вместо написания программ напрямую на JavaScript [3], а затем сравниваются два популярных в настоящее время проекта для генерации кода JavaScript из OCaml: BuckleScript[1] [1] и Js_of_ocaml [4]. Также обосновавается выбор языка Lua в роли целевого.

3.1. JavaScript

Несмотря на преимущества JavaScript, написание программ на данном языке сопряжено с рядом аспектов, которые многие программисты причисляют к недостаткам. Далее рассмотрены основные из них:

- Ненадёжность: слабая динамическая типизация допускает ошибки времени исполнения, многие из которых можно исключить средствами статической проверки типов в OCaml
- Наличие недостижимого кода: множественные зависимости от модулей порождают большое количество кода, который на самом деле не используется в проекте, что в свою очередь замедляет загрузку, парсинг и интерпретацию. Оба BuckleScript[2] и Js_of_ocaml [7] обеспечивает исключение недостижимого кода из скомпилированной программы
- JavaScript является динамическим языком программирования, из-за чего оптимизации кода, производимые JIT-компилятором, могут негативно сказываться на производительности программ во время их исполнения [6]. BuckleScript производит на выходе код, предрасположенный[3] к срабатыванию эвристик оптимизации JavaScript, тем самым повышая эффективность исполняемой программы

[1] Переименован в ReScript в июле 2020

[2] https://rescript-lang.org/docs/manual/latest/introduction#high-quality-dead-code-elimination

[3] https://rescript-lang.org/docs/manual/latest/introduction#readable-output--great-interop

3.2. Js_of_ocaml

Данная реализация использует низкоуровневый байткод OCaml для компиляции в JavaScript. По той причине, что API виртуальной машины OCaml меняется значительно реже самого языка, в проекте Js_of_ocaml легче поддерживать совместимость с актуальной версией OCaml. Также поддерживается раздельная компиляция, позволяющая переиспользовать однажды скомпилированные модули.

К минусам данного подхода относят нечитаемый код на выходе – в JavaScript-сообществе возможность переиспользовать существующие решения высоко ценится, однако видимая непонятность кода может препятствовать попыткам встроить его в новую программу. Эта же причина затрудняет отладку кода после компиляции в JavaScript, что усложняет процесс разработки и делает Js_of_ocaml недружелюбным к новичкам.

Помимо вышеперечисленного, Js_of_ocaml на момент написания статьи не поддерживает[4] следующие детали OCaml:

- Слабые ссылки
- Большая часть модуля Sys
- Некоторые библиотеки, поставляемые с OCaml
- Оптимизация хвостовой рекурсии в общем случае

3.3. BuckleScript

В отличие от Js_of_ocaml, BuckleScript транслирует исходный код на OCaml в читаемый код на JavaScript. Стратегическое решение осуществлять трансляцию из исходного кода приводит к сложностям при поддержке обновлений языка, в следствие чего иногда может возникать отставание поддерживаемой версии OCaml в BuckleScript от актуальной версии OCaml. Как и в Js_of_ocaml, имеется поддержка раздельной компиляции.

[4] https://ocsigen.org/js_of_ocaml/3.5.1/manual/overview

К отличительным чертам BuckleScript относится полная поддержка языка OCaml, благодаря которой ограничение на добавление зависимостей снимается: проекты, созданные с помощью BuckleScript, могут использовать почти полную мощь многочисленных библиотек как из экосистемы OCaml, так и JavaScript; а в силу понятности кода, он также может легко встраиваться в сторонние проекты.

В качестве дополнительного преимущества BuckleScript можно отметить большое и активное сообщество, которое разрабатывает вспомогательный API для удобства пользователей и поддерживает систему сборки, время работы которой на пару порядков быстрее[5] альтернатив.

3.4. Lua

Выбор языков при разработке для встроенных систем достаточно широк, однако у многих из них исторически сложились свои области применения. Так, Lua сегодня используется в программах компьютерной вёрстки, пакетных менеджерах, игровых движках и портативных устройствах. Язык был спроектирован компактным и простым в смысле минимальности базовых синтаксических средств и одновременно удобным в освоении для не профессиональных программистов. Так же Lua имеет эффективные средства межъязыкового взаимодействия, ориентированные, главным образом, на вызов библиотек C и на работу в C-окружении.

Компактность языка даёт преимущество перед JavaScript не только в легкости изучения, но и в более важной для аппаратной части стороне: размер библиотек, необходимый для запуска приложений, может варьироваться от 40КБ при исключении из зависимостей всех сторонних библиотек до 150-200КБ при добавлении встраиваемого интерпретатора — в то же время встраиваемые системы для JavaScript (например, *Ducktape*[6]) занимают 100-400КБ.

[5] `https://rescript-lang.org/docs/manual/latest/build-performance`
[6] `https://duktape.org/`

4. Реализация

С исходным кодом проекта можно ознакомиться на GitHub репозитории[7]. Далее в этой главе описываются принятые решения по выбору архитектуры, используемых технологий и некоторые детали реализации.

4.1. Архитектура

По завершению этапа исследования архитектура BuckleScript была выбрана в качестве опорной при реализации проекта в силу своих следующих преимуществ:

1. Трансляция из исходного кода, а не байт-кода, благодаря чему становится удобнее производить более читаемый и абстрагированный код вместо низкоуровневых инструкций

2. Возможность переиспользовать некоторые из модулей, созданных для инфраструктуры проекта

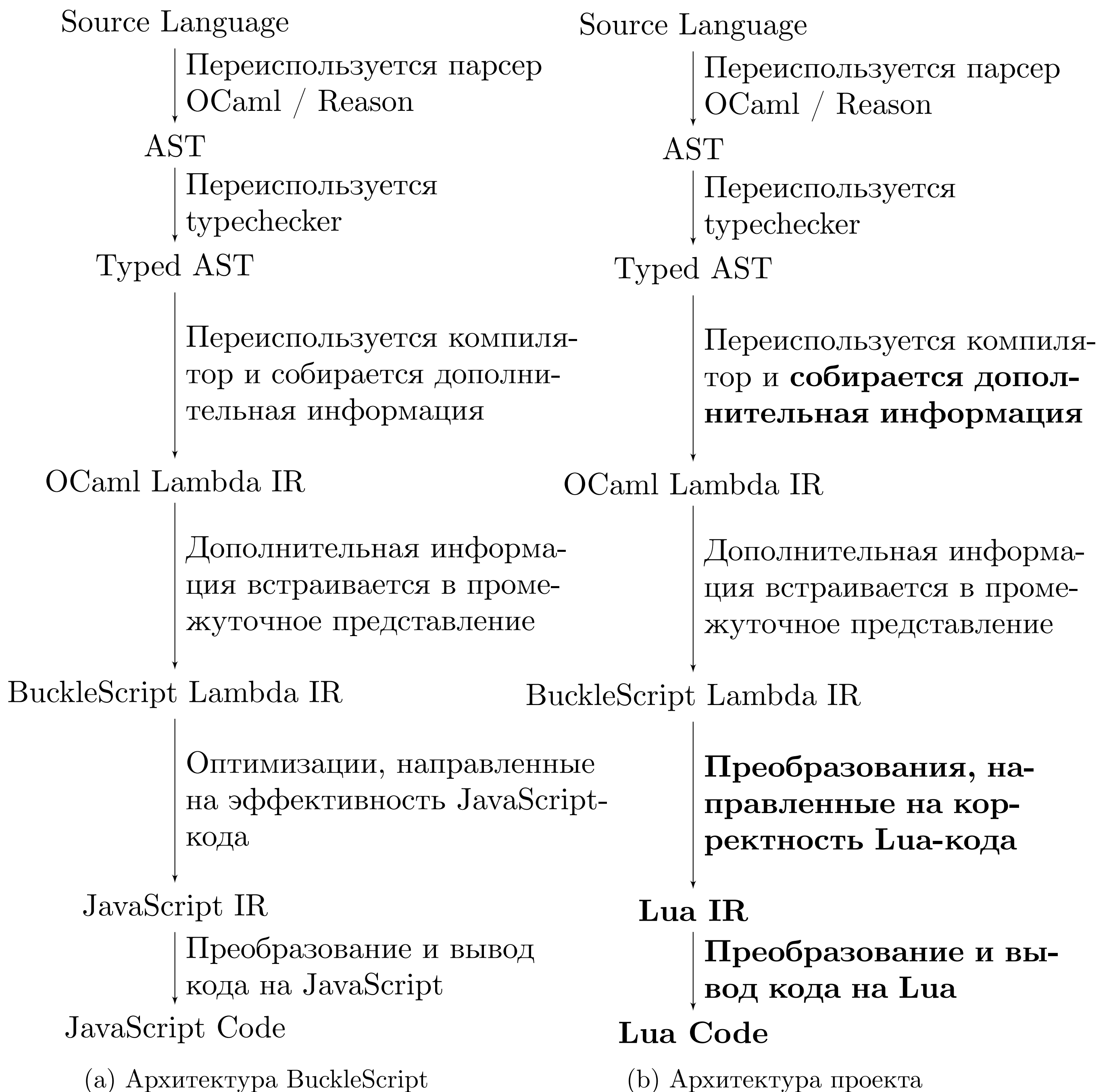
В BuckleScript целевым языком является JavaScript, поэтому, переиспользуя начальные этапы трансляции, необходимо заменить вывод на синтаксис Lua, переработав основные управляющие конструкции и интерфейсы. На нижеприведённом рис. 1 можно изучить изменения в высокоуровневой архитектуре проекта, которые были внесены в ходе практической работы.

4.2. Стек технологий

Выбор технологий обусловлен с одной стороны совместимостью с BuckleScript (из-за чего основным языком разработки является OCaml и используется система сборки ninja[8]), с другой – выбором целевого языка (имеются вставки кода на Lua).

[7]https://github.com/Tsirikov/bucklescript/tree/luacaml

[8]https://ninja-build.org/



(a) Архитектура BuckleScript

(b) Архитектура проекта

Рис. 1: Адаптация архитектуры для задач проекта

4.3. Детали реализации

В данной секции сначала перечислена краткая выборка из представляющих интерес проблем, решённых при реализации проекта, в хронологическом порядке написания соответствующего кода, а затем описано поддерживаемое подмножество языка OCaml:

1. Конкатенация строк: в JavaScript для сложения численных типов и

конкатенации строк используется общий оператор + из соображений о наличии неявного приведения типов. С учётом этого в BuckleScript при сборе дополнительной информации о типах на этапе перехода от "Typed AST" к "OCaml Lambda IR" отсутствовала необходимость в добавлении меток, позволяющих отличать строковые значения. Однако в Lua для конкатенации строк используется оператор `..`, в то время как применение оператора + к значениям строкового типа приводит к завершению программ с ошибкой. Поэтому на этапе сбора информации было добавлено сохранение метки строк, а в при выводе кода, полученного в ходе работы программы, стали использоваться дополнительные сопоставления с образцом

2. Оператор длины для строк и таблиц: строки и массивы в JavaScript являются экземплярами классов, у которых определёно свойство **`length`** целочисленного типа, представляющее собой информацию о длине соответствующей структуры. В Lua используется отличный подход к ООП, когда возможности классов реализуются через синтаксический механизм метатаблиц, и по умолчанию строки и таблицы не имеют поля **`length`** – для этой цели используется унарный оператор **`#`**, возвращающий количество байт в строке (что равнозначно количеству символов в ней, так как каждый символ кодируется одним байтом) или первый индекс из множества натуральных чисел, такой, что соответствующий элемент таблицы не определён (т.е. имеет значение **`nil`**), уменьшенный на единицу. Для решения этой проблемы так же, как и для предыдущего пункта, потребовалось обработать дополнительные случаи сопоставления с образцом при обработке обращения к полю класса

3. Представление тернарного оператора **`?:`**: в Lua отсутствует тернарный оператор **`?:`**, однако его легко выразить через операторы логических связок благодаря следующим обстоятельствам: первое заключается в том, что лишь два значения трактуются как "ложь" – **`false`** и **`nil`**, все остальные – как **`true`**, а второе, что операторы **`or`** и **`and`** возвращают не логическое значение, а первый или вто-

рой аргумент. Таким образом, выражение `a` **`and`** `b` **`or`** `c` выполняет в точности ту же функцию, что и **`?:`**, т.е. возвращает `b`, когда `a` интерпретируется как истина, и `c` иначе

4. Области видимости и связывание идентификаторов: оба языка JavaScript и Lua реализуют лексическое связывание, и конструкция **`let`** `x` равнозначна **`local`** `x` при правильном определении блока, на который распространяется её действие

5. Сопоставление с образцом: при трансляции в JavaScript эта конструкция выражалась через **`switch-case`**, однако в Lua подобная управляющая структура отсутствует, из-за чего возникла необходимость представить её в виде блоков **`if-then-elif-else`** так, чтобы поддерживалась концепция "falls through". Это было достигнуто через подходящее определение уровней вложенности для ветвлений

6. Обработка исключений: отличием Lua от JavaScript в плане обработки ошибок является то, что вместо блоков выражений внутри конструкции **`try-catch`** используется функция **`pxcall`**(**`mayraise`**, **`handler`**), которая первым аргументом принимает функцию, которая не принимает аргументов и может произвести исключение, а вторым – функцию, которая принимает единственный аргумент – ошибку – и запускается, если первая завершилась аварийно. Создать такие функции средствами Lua можно через анонимные функции, телами которых являются соответствующие блоки

7. Импорт: чтобы импортировать модуль в Lua, в нём явным образом должно быть указано, что этот модуль доступен для импорта, и перечислены все поля, которые будут видны в вызывающем модуле. Для этого на этапе перехода от "Typed AST" к "OCaml Lambda IR" собиралась дополнительная информация о том, что необходимо предоставить для импорта, и затем в конце модуля эти данные объявлялись по правилам Lua

8. Объявление пользовательских типов данных: с помощью механиз-

ма хэш-таблиц для структур было зарезервировано поле, в котором хранилась информация о типе, и поля для данных, которые благодаря динамической типизации походили для всех возможных аргументов, которые могли быть переданы в конструктор типа

9. Каррирование было выражено средством замыкания функций

В итоге поддерживаются следующие конструкции языка OCaml:

1. Арифметические, битовые, булевы и строковые операции, операции сравнения
2. Управляющие конструкции: цикл, ветвление (в том числе сопоставление с образцом)
3. Функции, рекурсия, каррирование
4. Обработка исключений
5. Неформатированный вывод
6. Импорт модулей

Нижеперечисленные конструкции OCaml не поддержаны:

1. Массивы, списки
2. Функции ввода
3. Форматирование строк
4. Взаимодействие с ОС

4.4. Тестирование

В проекте средствами системы сборки ninja настроено регрессионное тестирование, заключающееся в применении программы-транслятора после каждой сборки к набору тестовых файлов, находящихся в директории

"/jscomp/test" – по завершению работы которой делался вывод о способности программы завершаться с нулевым кодом ошибки на множестве входных данных.

Для проверки же корректности получаемых программ осуществлялось ручное тестирование избранных программ, приведённых в директории "/lib/js" под именами, начинающимися с "_test_" (среди программ, например, находятся применение каррирования, объявление, создание и сопоставление с образцом пользовательского типа данных, вычисление чисел Фиббоначи различными способами). Они также транслирвоались из OCaml в Lua, а затем запускались с различными входными данными.

5. Заключение

В ходе выполнения учебной практики были достигнуты следующие результаты:

1. Проведено исследование предметной области, в том числе изучены решения, принятые в аналогах проекта

2. Спроектирована трансляция структур данных из OCaml в Lua

3. Реализован прототип инструмента

4. Полученное решение протестировано

Реализация нижеперечисленной функциональности не ставилась задачей текущей работы, но её добавление в будущем может улучшить качество, скорость и удобство применения программ, полученных с помощью разработанного инструмента:

1. Использование библиотек Lua для оптимизации работы со строками, массивами, таблицами, математическими функциями, поддержки UTF-8 и взаимодействия с операционной системой

2. Использование сопрограмм

3. Внедрение C FFI, поддерживаемое Lua

Список литературы

[1] BuckleScript. — https://github.com/bucklescript/bucklescript.

[2] Ierusalimschy Roberto, de Figueiredo Luiz Henrique, Celes Waldemar. Lua 5.3 Reference Manual. — 2018. — Access mode: https://www.lua.org/manual/5.3/.

[3] International ECMA. ECMAScript® 2019 Language Specification. — 10 edition. — 2019. — June. — Access mode: http://www.ecma-international.org/ecma-262/10.0/index.html.

[4] Js_of_ocaml. — https://github.com/ocsigen/js_of_ocaml.

[5] The OCaml system release 4.06: Documentation and user's manual / Xavier Leroy, Damien Doligez, Alain Frisch et al. — 2017. — Access mode: http://caml.inria.fr/pub/distrib/ocaml-4.06/ocaml-4.06-refman.pdf.

[6] St-Amour Vincent, yu Guo Shu. Optimization Coaching for JavaScript // 29th European Conference on Object-Oriented Programming (ECOOP 2015) / Ed. by John Tang Boyland. — Vol. 37 of Leibniz International Proceedings in Informatics (LIPIcs). — Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. — P. 271–295. — Access mode: http://drops.dagstuhl.de/opus/volltexte/2015/5226.

[7] Vouillon Jérôme, Balat Vincent. From bytecode to JavaScript: the Js_of_ocaml compiler // Softw., Pract. Exper. — 2014. — Vol. 44, no. 8. — P. 951–972. — Access mode: https://doi.org/10.1002/spe.2187.