Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Перенос компилятора OCaml в инфраструктуру LLVM

Курсовая работа студента 344 группы
Терешина Романа Юрьевича

Научный руководитель ..... к.ф.-м.н. Д. Ю. Булычев

Санкт-Петербург
2014

Содержание

Введение

Разработка компилятора для современного языка программирования общего назначения — большая и сложная задача. Один из методов упрощения этой задачи состоит в создании и использовании инфраструктур компиляции. Успешность такого подхода связана с тем, что несмотря на многообразие языков программирования и целевых платформ, при разработке очередного компилятора почти всегда можно что-то переиспользовать из существующих. Классический приём заключается во введении некоторого универсального промежуточного представления, которое с одной стороны способны генерировать синтаксические анализаторы для различных входных языков, а с другой — принимать в качестве входного языка генераторы машинного кода для всевозможных целевых платформ. Таким образом удаётся сократить число разрабатываемых компонент с $n \times m$ до $n+m$, где n — число входных языков, а m — число платформ. Как правило, инфраструктура компиляции включает также целый ряд инструментов, осуществляющих некоторые преобразования промежуточного представления, например, с целью оптимизации. Среди существующих инфраструктур наиболее широким спектром поддерживаемых аппаратных архитектур и высокой надёжностью отличается GCC (GNU Compiler Collection) [13, 10], ориентацией на исследования в области оптимизации кода для высокопроизводительных платформ — проекты SUIF [8] и Zephyr[1]. В рамках этой работы внимание будет приковано к проекту LLVM [6] (в прошлом Low Level Virtual Machine, на настоящий момент неделимое имя, представляющее целое семейство связанных проектов).

Инфраструктура LLVM — это относительно устойчивая спецификация промежуточного представления, называемого LLVM IR (Intermediate Representation) [7], и большое количество программ, предназначенных для работы с ним. Среди них центральное место занимает модульный, многопроходной оптимизатор. Его дополняют кодогенераторы для наиболее популярных платформ, входящие в состав как статического компилятора, так и динамического JIT-компилятора, оптимизирующий редактор связей, отладчик, библиотеки времени исполнения для некоторых языков и другие инструменты. Основной язык реализации LLVM — C++. Дизайн LLVM IR преследует две основные цели: во-первых, позволить этому представлению быть одновременно независимым от входного языка и достаточно выразительным, чтобы успешно вмещать все его абстракции и во-вторых, сохранить любую информацию, которая может быть полезна оптимизатору. Однако множество фактически поддерживаемых LLVM языков не очень велико. Благодаря переиспользованию компонент инфраструктуры GCC и проекту Clang[2] оно включает C, C++, Objective-C, D, Fortran. Все эти языки объединяет набор свойств: строгая, статическая типизация, ручное управление памятью, недостаточно развитые и/или эффективные механизмы обработки ошибок и исключительных ситуаций, поддержка преимущественно императивных и объектно-ориентированных конструкций. Существует опыт реализации функциональных языков для LLVM, но его лишь с некоторыми оговорками можно признать удач-

[1] http://www.zephyr-software.com

[2] http://clang.llvm.org

ным [2, 3, 5]. Таким образом, возможность и простота реализации в среде LLVM языков с развитой поддержкой времени исполнения, включающей автоматическое управление памятью и сборку мусора, нерасточительные механизмы генерации и обработки исключений, а также функциональных языков с присущими им свойствами вызывает сомнения и требует проверки. Ещё больший интерес представляет возможность такой реализации ценой умеренных усилий и со значительным переиспользованием кода существующих компиляторов для соответствующих языков. Подобная апробация LLVM и составляет фокус этой работы.

В качестве модельного языка, обладающего всеми перечисленными выше свойствами, было использовано некоторое подмножество языка OCaml[3] [14, 15]. Он представляет собой диалект языка ML, сочетающий функциональные, императивные и объектно-ориентированные конструкции. К его достоинствам традиционно относят статическую типизацию и автоматический вывод типов, параметрический полиморфизм, сборку мусора, развитую систему модулей. Для OCaml на текущий момент существует ровно один компилятор в машинный код (но для нескольких популярных платформ): ocamlopt. Он реализует лишь небольшое количество известных методов оптимизации кода, потому переиспользование оптимизатора LLVM может иметь собственную ценность для OCaml. Компилятор ocamlopt написан на OCaml и имеет в части, интересной с точки зрения этой работы, умеренную сложность, что обещает возможность достаточно быстрой его модификации. Удобно, что LLVM обладает интерфейсом для доступа к некоторым его инструментам из программ на языке OCaml.

Отдельный интерес представляет реализация сборщика мусора для LLVM. Эта задача выходит за рамки данной работы, однако версия компилятора ocamlopt для LLVM может быть использована в качестве тестовой площадки для разработки, отладки, тестирования и исследования свойств сборщиков мусора.

Таким образом, цель данной работы — перенос компилятора ocamlopt в инфраструктуру LLVM с целью изучения осуществимости такой реализации, выявления возможных проблем LLVM в контексте подобных задач, и, по возможности, их решения.

[3] http://caml.inria.fr

1 Обзор связанных работ

К настоящему времени уже было сделано несколько попыток использования LLVM в качестве бэкенда в компиляторах для функциональных языков программирования. Перед обсуждением этих работ стоит несколько расширить описание LLVM, приведённое во введении, а именно дать характеристику LLVM IR.

LLVM IR — это строго типизированная RISC-подобная SSA-форма (Static Single Assignment Form [1, 12]), являющаяся универсальным промежуточным представлением программ в инфраструктуре LLVM. Система типов LLVM включает большое количество целочисленных типов практически произвольной битовой ширины, несколько числовых типов с плавающей точкой, векторные числовые типы, локальные метки, множество конструкторов типов со структурным равенством на них. Конструкторы типов позволяют определять структуры, массивы, указатели и функциональные типы произвольного уровня вложенности. В LLVM IR адресная арифметика полностью абстрагирована высокоуровневой операцией вычисления адреса элемента сложного типа. Большинство операций в LLVM IR трехадресные, их аргументами могут быть исключительно типизированные виртуальные регистры. Значением регистра может быть указатель на некоторую область в памяти: локальную для текущей функции или глобальную. Набор допустимых операций с памятью невелик, состоит из операций чтения значения по некоторому адресу и записи значения по адресу. Одна из инструкций LLVM IR позволяет выделять участок памяти в записи активации функции аналогично функции alloca библиотеки glibc[4]. В силу использования SSA-формы, LLVM IR допускает лишь одно определение значения каждого виртуального регистра, число которых неограничено. Для представления нескольких возможных значений под именем одного виртуального регистра SSA-форма вводит понятие ϕ-функции, которая позволяет связать регистр с тем или иным значением в зависимости от пути, по которому управление пришло в текущую точку программы. LLVM IR полностью абстрагирует вызов функции, запись активации функции и соглашения о вызовах и требует явного представления функции в виде линейной записи ориентированного графа потока управления, каждый узел которого представляет собой именованный некоторой меткой базовый блок. LLVM IR содержит большое количество различных аннотаций и встроенный язык описания метаданных, которые позволяют передавать от одной стадии компиляции к другой (в том числе от одного прохода оптимизатора к другому) самую разную информацию. Также LLVM IR допускает расширение с помощью встраиваемых функций (intrinsic functions), с использованием которых в LLVM реализованы интерфейс к механизму обработки исключений, являющийся частью Intel Itanium C++ ABI[5], и интерфейс ко внешнему сборщику мусора. Альтернативный способ реализации исключений для LLVM заключается в использовании функций setjmp/longjmp стандартной библиотеки C [4] или соответствующих им встраиваемых функций. Кроме генерации исключения с последующим его перехватом в LLVM IR нет способа передать управление из одной

[4]http://www.gnu.org/software/libc/libc.html

[5]http://refspecs.linuxbase.org/abi-eh-1.21.html

функции в другую, не используя стандартный механизм вызова функции и возвращения управления вызывающей функции.

1.1 LLVM-бэкенд для GHC (Haskell)

GHC (Glasgow Haskell Compiler[6]) — один из самых популярных компиляторов для языка Haskell. В рамках работы [3] для этого компилятора был успешно реализован LLVM-бэкенд, который согласно документации для GHC[7] готов к широкому использованию и в некоторых случаях генерирует код, существенно более производительный, чем код, генерируемый основным бэкендом GHC (прирост производительности в первую очередь касается программ, выполняющих большое количество арифметических операций). Его использование, однако, значительно увеличивает время работы самого компилятора. Реализованный бэкенд обладает полной бинарной совместимостью с остальными бэкендами GHC и опирается на немодифицированную библиотеку поддержки времени исполнения, в частности, работает с оригинальным сборщиком мусора GHC. Эта реализация не использует встроенную в LLVM поддержку сборки мусора, но осуществляет её самостоятельно. Для достижения текущего состояния LLVM-бэкенда потребовалось расширить LLVM специфическими соглашениями о вызовах (которые позволяют зафиксировать за частью машинных регистров специальное назначение и частично вывести их из-под контроля LLVM), а также решить некоторые другие нетривиальные задачи (например, организовать управление взаимным размещением машинного кода и данных в памяти, для которого LLVM не предоставляет никаких механизмов). Генерация LLVM IR здесь осуществляется в текстовой форме без использования инструментов LLVM.

1.2 LLVM-бэкенд для HiPE (Erlang OTP)

HiPE (High-Performance Erlang)[8] — это компилятор в машинный код для Erlang OTP (Open Telecom Platform)[9], для которого в рамках проекта ErLLVM[10] был разработан LLVM-бэкенд [5], полностью интегрированный в исполняющую систему Erlang OTP. Реализация этого проекта потребовала расширить LLVM Erlang-специфичными соглашениями о вызовах и усилить контроль за распределением регистров (образом, схожим с реализованным в [3]), а также модифицировать генератор прологов и эпилогов функций. Здесь использовалась встроенная в LLVM IR поддержка сборки мусора, что привело к снижению эффективности генерируемого кода. Эта поддержка обязывает хранить элементы корневого множества в записи активации функции (на стеке), а не в регистрах, и поддерживать эту память в актуальном состоянии во время выполнения программы. В связи с этим просходит увеличение размера записи активации функции и самого корневого множества,

[6] https://www.haskell.org/ghc/
[7] http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/code-generators.html
[8] http://www.it.uu.se/research/group/hipe/
[9] http://www.erlang.org/
[10] http://erllvm.softlab.ntua.gr/

а также числа операций чтения из памяти и записи в неё. Попытка уменьшить негативное влияние этих эффектов на производительность потребовала реализации анализа времени жизни элементов корневого множества, что усложнило LLVM-бэкенд, но полностью проблему не решило. Обработка исключительных ситуаций была реализована в соответствии с Intel Itanium C++ ABI. В большинстве случаев генерируемый новым бэкендом код имеет производительность, сравнимую с производительностью кода, генерируемого оригинальным компилятором HiPE. Основная причина невысокой производительности — неэффективная поддержка сборки мусора. Генерация LLVM IR здесь также осуществляется в текстовой форме, что является одной из причин меньшей скорости работы нового компилятора по сравнению с оригинальным. На текущий момент этот бэкенд является экспериментальным.

1.3 Проект ocamlllvm (OCaml)

ocamlllvm[11] — прототип LLVM-бэкенда для компилятора ocamlopt [2]. В этой работе была предпринята попытка реализовать для OCaml поддержку ещё одной целевой платформы — LLVM — со значительным переиспользованием кодогенератора и некоторых других модулей оригинального компилятора. Иначе говоря, эта работа рассматривает LLVM как реальную аппаратную платформу. Такой подход, в частности, позволил свести к минимуму время, необходимое для адаптации библиотеки поддержки времени исполнения. Эта реализация полагается на существующий в OCaml сборщик мусора. Однако, ввиду возникших трудностей, разработанный компилятор обладает рядом ограничений. Он не позволяет использовать оптимизатор LLVM, так как это нарушает корректность сборки мусора, генерирует медленно работающий код, не реализует поддержку оригинального механизма обработки исключений. Генерируемые ocamlllvm объектные модули не обладают бинарной совместимостью с модулями, генерируемыми оригинальным компилятором. Поддержка исключений в этой работе осуществлена с помощью механизма setjmp/longjmp, к недостаткам которого относятся потребление большого количества дополнительной памяти и ненулевые накладные расходы даже в случае исполнения программы без генерации исключительных ситуаций. Генерация LLVM IR здесь также осуществляется явным образом в текстовой форме.

[11] https://github.com/colinbenner/ocamlllvm

2 Реализация

Набор утилит, входящий в состав инфраструктуры OCaml, включает компиляторы ocamlc и ocamlopt, а также исполняемый модуль виртуальной машины ocamlrun. ocamlc является транслятором с языка OCaml в байт-код, который способна выполнять виртуальная машина OCaml. К достоинствам пары ocamlc/ocamlrun относятся высокая скорость компиляции, относительно низкая сложность реализации виртуальной машины для новой целевой платформы, мощная поддержка процесса отладки генерируемых программ. Основной её недостаток — невысокая скорость исполнения программ, скомпилированных в байт-код. Этот недостаток исправляет компилятор в машинный код ocamlopt, реализованный для таких архитектур, как AMD64, IA32, Power PC, ARM и SPARC[12]. Компиляция в обоих случаях представляет собой последовательность преобразований одного представления текста программы в другое. Компиляторы OCaml используют большое количество промежуточных форм, наиболее ранние из которых являются общими для ocamlc и ocamlopt. Оба компилятора осуществляют препроцессирование, лексический и синтаксический анализ исходного текста программы и преобразование его в нетипизированное абстрактное синтаксическое дерево (AST, Abstract Syntax Tree), вывод типов и проверку корректности программы с точки зрения системы типов с генерацией типизированного AST, компиляцию сопоставлений с образцом, устранение модулей и классов и другие преобразования с получением т.н. лямбда-представления, начиная с которого процессы компиляции в машинный код и в байт-код начинают различаться. Лямбда-представление ocamlopt преобразует в простую процедурную C-подобную форму, называемую Cmm. На этой стадии происходит преобразование функций-значений, в том числе замыканий, и статических данных программы в набор низкоуровневых определений функций и данных. Формы, следующие за Cmm, являются машинно-зависимыми.

2.1 Представление данных в OCaml

Перед тем, как перейти к описанию промежуточного представления, называемого Cmm, стоит сказать несколько слов о представлении данных в OCaml. В OCaml экземпляр любого типа, как встроенного, так и пользовательского, является значением (value). С точки зрения реализации языка, значение может быть либо целым числом, либо указателем. В обоих случаях оно занимает в точности одно машинное слово (шириной либо 32, либо 64 бита в зависимости от разрядности целевой платформы). Конкретное представление значения статическим свойством не является и в некоторых случаях может изменяться во время выполнения программы. Так, экземпляр целого, символьного или логического типа всегда является целочисленным значением, тогда как внутреннее представление кортежа — всегда указатель на блок (block), расположенный в динамической памяти (куче). Однако список (и любой другой вариантный тип) может быть как целочисленным значением (если он пуст или, в случае произвольного вариантного типа, не имеет аргументов),

[12] http://caml.inria.fr/ocaml/portability.en.html

так и указателем. OCaml позволяет оперировать как изменяемыми (mutable) значениями, так и неизменяемыми (immutable). Единственный способ создать изменяемое значение пользовательского типа — определить структуру (или объект) с mutable-полем, однако в языке существуют встроенные изменяемые типы, например, массив. Таким образом, если mutable-поле структуры или элемент массива имеет вариантный тип, например, является списком, его значение во время исполнения может быть как целым числом, так и указателем, что в большинстве случаев невозможно определить статически. Тем не менее, в любой момент времени о любом значении можно сказать, представлено оно целым числом или указателем. Такая возможность является необходимым условием осуществимости точной (неконсервативной) сборки мусора, которая реализована в OCaml. Любое целочисленное значение в OCaml хранится со сдвигом влево на один бит и установленным в 1 младшим битом, иначе говоря, в форме $2n + 1$, где n — представляемое целое число. Так как все указатели являются адресами блоков, расположенных в памяти выровнено по ширине машинного слова, их младшие биты равны нулю. Именно эти свойства и позволяют во время выполнения программы при необходимости различать целочисленные значения и указатели (по младшему биту значения). Однако такое представление увеличивает число машинных операций, необходимых для выполнения простейших арифметический действий. Например, в целочисленном случае сложение выполняется по формуле $(2n + 1) + (2m + 1) - 1 = 2(m + n) + 1$, где $2n + 1$, $2m + 1$ и $2(m + n) + 1$ — представления операндов и результата соответственно, и потому требует двух операций, а не одной. Способ хранения значений с плавающей точкой также имеет свои особенности. Отображение арифметических действий в наборы машинных операций в компиляторе ocamlopt осуществляется достаточно рано и Cmm оперирует уже машинными представлениями всех значений.

2.2 Уровень представления Cmm

Cmm — это одна из многочисленных частичных реализаций промежуточного представления, называемого C−− [11, 9]. C−− представляет собой подмножество языка C с машинно-ориентированной системой типов, гибким языком описания данных, средствами контроля за размещением данных и машинного кода в памяти, интерфейсом к библиотеке поддержки времени исполнения, в том числе к сборщику мусора и механизму обработки исключительных ситуаций. В неизменном виде и вместе с оригинальной библиотекой поддержки времени исполнения большой популярностью C−− пользоваться не стал, однако достаточно часто в качестве одного из промежуточных представлений встречаются его модификации [3]. Далее в тексте под Cmm следует понимать реализацию C−− в компиляторе ocamlopt.

Cmm, в отличие от C−−, имеющего богатый набор типов, оперирует лишь несколькими машинными типами, из которых широко используются три: машинное слово для представления значений, вещественный тип, соответствующий типу double в языке C, однобайтовый целочисленный тип для представления символов и строк. Cmm ни в какой

форме не использует встроенную в C−− поддержку исключений и сборки мусора, но в то же время содержит специфические для OCaml конструкции. Cmm-представление — это последовательность определений непрерывных блоков статических данных и функций текущей единицы компиляции, следующих в произвольном порядке. Описание каждого блока данных в Cmm — это список директив, каждая из которых может быть одного из трёх видов:

- размещение конкретного значения того или иного машинного типа в памяти непосредственно за предыдущим, если такое есть;
- создание указателя на последующее значение и связывание его либо с меткой, доступной только локально в рамках текущего определения блока данных, либо с глобально доступным символом;
- выравнивание по некоторой границе.

Также с помощью определений статических данных модуль экспортирует набор символов (именующих как данные, так и функции), которые будут доступны из других модулей. Определение функции включает в себя её имя, список имён формальных параметров, выражение, являющееся телом функции и некоторую отладочную информацию. Выражение в Cmm может включать простейшие операции чтения из памяти и записи в неё по указателю, а также оператор присваивания переменной-счётчику цикла некоторого значения, арифметические операции, оперирующие как целочисленными значениями (в том числе адресами), так и значениями с плавающей точкой, определения различных констант, ссылки по имени на переменные, let-выражения, различные операторы, контролирующие поток управления (циклы, переходы по меткам, условные операторы и switch-операторы), вызовы функций. Из специфических конструкций следует назвать операторы контроля выхода за границы массива, генерации и перехвата исключений. Сборка мусора совершенно прозрачна для Cmm и не имеет на его уровне никакого представления.

Cmm существенным образом опирается на поддержку времени исполнения. Соответствующая библиотека реализована на языке C и языках ассемблеров для целевых платформ.

2.3 Общие замечания о реализации

Среди используемых компилятором ocamlopt промежуточных представлений к LLVM IR наиболее близок именно Cmm. Представления, следующие за Cmm, являются машинно-зависимыми, а преобразования, выполняемые над ними, решают задачи, полностью реализованные в LLVM: распределение и назначение регистров, выбор команд и прочее. Поэтому их использование в качестве отправной точки нецелесообразно. Тем не менее, даже Cmm содержит некоторые OCaml-специфичные конструкции, а в предшествующих представлениях их число только выше. Поэтому именно Cmm стал источником трансляции в LLVM IR. Иначе говоря, в данной работе был реализован транслятор из Cmm в LLVM IR.

Генерируемый им код опирается на библиотеку поддержки времени исполнения на языке C. С целью упрощения задачи было решено использовать на уровне LLVM IR соглашения о вызовах языка C, что является поведением LLVM по умолчанию. Оригинальный компилятор ocamlopt использует собственные, несовместимые соглашения о вызовах. Этот факт является одной из причин двоичной несовместимости объектных модулей, генерируемых компиляторами ocamlopt и ocamlnc (здесь и далее компилятор, реализованный в рамках данной курсовой работы, будет именоваться именно так). Ввиду значительной сложности оригинальной библиотеки поддержки времени исполнения было решено строить собственную библиотеку и наращивать её функциональные возможности по мере расширения поддерживаемого подмножества OCaml, максимально переиспользуя существующую библиотеку. Такой подход в том числе позволяет последовательно расширять набор тестов, с помощью которых осуществляется контроль корректности реализации.

Транслятор Cmm-представления в LLVM IR был реализован на языке OCaml. Генерация LLVM IR этим транслятором осуществляется при значительной поддержке функций, входящих в состав интерфейса доступа к инструментам LLVM из языка OCaml, что позволяет несколько сократить размер модуля, повысить надёжность реализации, её устойчивость к возможным изменениям в LLVM в следующих его версиях и скорость работы компилятора.

Оригинальный компилятор был дополнен модулем, осуществляющим указанную трансляцию, набор распознаваевых им аргументов командной строки расширен ключом, включающим вывод LLVM IR в стандартный поток вывода. Оптимизация и компиляция сгенерированного LLVM IR выполняется инструментами opt и llc[13] соответственно. Сборка библиотеки поддержки времени исполнения возможна любым компилятором C, поддерживающим стандарт C99, исполняемого файла программы — соответствующим редактором связей.

Важное общее замечание по реализации заключается в том, что для компиляторов OCaml существует только пользовательская документация. Никакой внутренней документации для разработчиков нет. Поэтому основным источником информации о назначении и контрактах тех или иных конструкций представления Cmm и о логике компилятора служит сам код компилятора. Ввиду того, что исследовать ту или иную конструкцию Cmm значительно проще с тестовыми примерами на языке OCaml, способными заставить компилятор её сгенерировать, чем без таких примеров, в реализации транслятора из Cmm в LLVM IR на текущий момент можно встретить нереализованные конструкции. В большинстве случаев это означает, что назначение и способы использования этих конструкций понять не удалось ввиду безуспешности попыток придумать тест на языке OCaml, который приводил бы к их генерации.

[13]opt — оптимизатор LLVM, llc — статический компилятор LLVM IR

2.4 Представление данных в LLVM IR

Как уже было указано, LLVM IR — типизированное представление. Известно, что часть оптимизаций, реализованных в LLVM, использует информацию о типах. Например, авторы [6] отмечают, что значительные усилия при реализации трансляции низкоуровневого нетипизированного промежуточного представления GCC RTL [10] в LLVM IR были приложены к восстановлению информации о типах, в том числе с использованием отладачной информации, генерируемой GCC. Компилятор ocamlopt удаляет информацию о типах из хранимого представления программы на ранних стадиях компиляции. Тем не менее в связи с особенностями представления значений в OCaml, которые были кратко описаны в обзорной части, от вывода типов Cmm-значений было решено отказаться. Большая часть значений, которыми оперирует Cmm, либо являются 64-битными целыми числами, либо могут таковыми оказаться во время выполнения. В связи с тем, что LLVM IR — статически и строго типизированное представление, приходится все такие значения хранить в некотором унифицированном виде (как 64-битное целое LLVM IR — тип i64), и выполнять явное преобразование типов при каждом использовании значения. Свободу манипулирования значениями также ограничивает следующее требование: все виртуальные регистры LLVM, которые используются как аргументы ϕ-функции, должны иметь один тип. Всё это существенно усложняет вывод типов для полиморфного кода. Далее в тексте предполагается, что все необходимые явные преобразования типов осуществляются верно и манипуляциям с типами внимание уделяться не будет.

В целом, компилятор ocamlnc полностью сохраняет оригинальное представление блоков OCaml, а значит и кортежей, вариантов, замыканий и прочих конструкций.

2.5 Компиляция статических данных

Статическими данными программы принято называть данные, традиционно помещаемые компиляторами в секции .data и .bss объектных и исполняемых модулей. В языке C им соответствуют глобальные и статические переменные. Компиляторы OCaml таким образом размещают в модулях различные глобальные переменные, а также константы (обычно входящие в состав программы как литералы), в том числе сложно устроенных типов: списков, деревьев и т.д. Каждому блоку данных Cmm транслятор ставит в соответствие значение структурного типа в LLVM, поскольку так можно гарантировать непрерывное следование друг за другом в памяти элементов блока данных. Каждому элементу блока данных Cmm ставится в соответствие неименованное поле структуры LLVM IR. Так как часть элементов является указателями на метки или символы, которые могут быть ещё не определены в программе, компиляция статических данных осуществляется в два прохода. Первый проход формирует несколько таблиц, ставящих в соответствие определённым меткам и символам имя LLVM-структуры и номер поля в ней, а также запоминает директивы на размещение указателей на символы и метки. Второй проход, который является самым последним этапом работы транслятора, исполняет эти директивы в контексте, в

котором уже гарантировано определены все необходимые символы и метки. Тот факт, что компилятор ocamlopt до генерации Cmm-представления выполняет множество проверок на корректность программы и во многих других случаях позволяет делать удобные предположения о свойствах Cmm-представления.

2.6 Трансляция выражений

Определения констант и арифметические операции транслируются в LLVM IR прямолинейным образом. Перенос вещественных операций сравнения из Cmm в LLVM IR требует аккуратного обращения со значениями NaN[14]. Операции чтения и записи по некоторому смещению не нашли прямого отображения, так как Cmm использует явную адресную арифметику, которой нет в LLVM IR. По этой причине в Cmm-представлении выполняется поиск типичных фрагментов, выполняющих операции с указателями, и их замена на эквивалентные конструкции LLVM IR с использованием инструкции getelementptr. Трансляция имен переменных и других символов требует поддержания нескольких таблиц символов.

2.7 Трансляция операторов, модифицирующих поток управления

Трансляция даже простых императивных конструкций в SSA-форму может быть нетривиальной. Сложности возникают при попытке модифицировать какие-либо перменные, например, счётчики циклов. Для того, чтобы пользователи LLVM не были вынуждены реализовывать полный набор алгоритмов конструирования SSA, разработчики LLVM рекомендуют использовать описанный далее приём. Память под изменяемые переменные можно выделить на стеке (в записи активации текущей функции) и при каждой операции чтения значения или его модификации выполнять соответствующую операцию с памятью по указателю. Один из первых проходов оптимизатора LLVM (mem2reg) способен распознавать подобные шаблоны использования переменных на стеке и переносить большинство таких переменных в виртуальные регистры, при необходимости генерируя новые базовые блоки и вызовы ϕ-функции. Оптимизация mem2reg требует, чтобы выделение памяти под такие переменные осуществлялось в первом базовом блоке функции. Выполнение такого требования несколько усложняет реализацию транслятора и более того, с помощью это приёма не удаётся полностью избавиться от “ручного” генерирования SSA-формы. Целый ряд операторов Cmm, модифицирующих поток управления, использует некоторый аналог безусловного перехода на метку. Часто такая метка соответствует коду, обрабатывающему какую-нибудь нетипичную ситуацию вроде неудачного сопоставления с образцом, и переходы на неё могут осуществляться из самых разных базовых блоков функции. Корректное конструирование графа потока управления функции и генерация подходящих вызовов ϕ-функции требует некоторого внимания. Ещё одним примером (помимо адресной арифметики), служащим в пользу того факта, что LLVM IR по набору поддерживаемых

[14]Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture

конструкций находится уровнем выше Cmm, служит switch-оператор. Его LLVM-версия не требует, чтобы целочисленные метки оператора образовывали непрерывный целочисленный отрезок, но предполагает явное определение перехода по умолчанию (аналогичного метке default в языке C). Cmm-версия switch-оператора может работать только с непрерывным набором меток от нуля до некоторого значения и не имеет ветки по умолчанию. К необходимому виду switch-операторы преобразуются компилятором ocamlopt на ранних стадиях трансляции.

2.8 Поддержка замыканий и частично применённых функций

В OCaml есть поддержка функций высшего порядка, частичного применения функций и замыканий. Каждое функциональное значение представляет собой односвязный список блоков. Последний блок содержит указатель на код представляемой этим значением функции f от N аргументов (см. рисунок 1) и значения ($x_1,\ x_2,\ \ldots,\ x_E$), которые используются в теле функции f, но не являются её параметрами или локальными переменными. Каждый блок, кроме последнего, содержит один аргумент arg_i, к которому представляемая функция уже была частично применена. Любой фрагмент такого списка, начинающийся с некоторого блока и завершающийся последним блоком, является функцией, частично применённой к M аргументам. Каждый блок содержит также число k аргументов, которые ещё не были связаны, и указатели на одну или две вспомогательные функции. К ним относятся функция частичного применения $caml\_curryN\_M$ и функция полного применения $caml\_curryN\_M\_app$. Первая осуществляет связывание ещё одного (очередного) аргумента функции f, вторая — реальный вызов функции f и требует наличия всех k несвязанных аргументов. На уровне Cmm применение (частичное или полное) функционального значения к K аргументам осуществляется вызовом вспомогательной функции $caml\_applyK$, которая в зависимости от значения k выполняет либо K частичных применений, либо полное применение функции. Устройство замыканий было сохранено неизменным, для чего транслятор был дополнен генераторами всех необходимых вспомогательных функций на LLVM IR.

2.9 Реализация исключений

LLVM поддерживает два различных механизма обработки исключительных ситуаций: с помощью нелокальных переходов setjmp/longjmp и табличным методом в соответствии со спецификацией Intel Itanium C++ ABI. Первый может быть реализован с использованием либо встроенных функций llvm.eh.sjlj.setjmp и llvm.eh.sjlj.longjmp, либо функций setjmp/longjmp стандартной библиотеки языка C. Встроенные функции llvm.eh.sjlj.* плохо документированы, а разработчиками LLVM рекомендован к использованию только табличный механизм. В связи с этим была выполнена экспериментальная реализация исключений с помощью функций setjmp/longjmp стандартной библиотеки C, которая впоследствии была замещена реализацией табличного подхода.

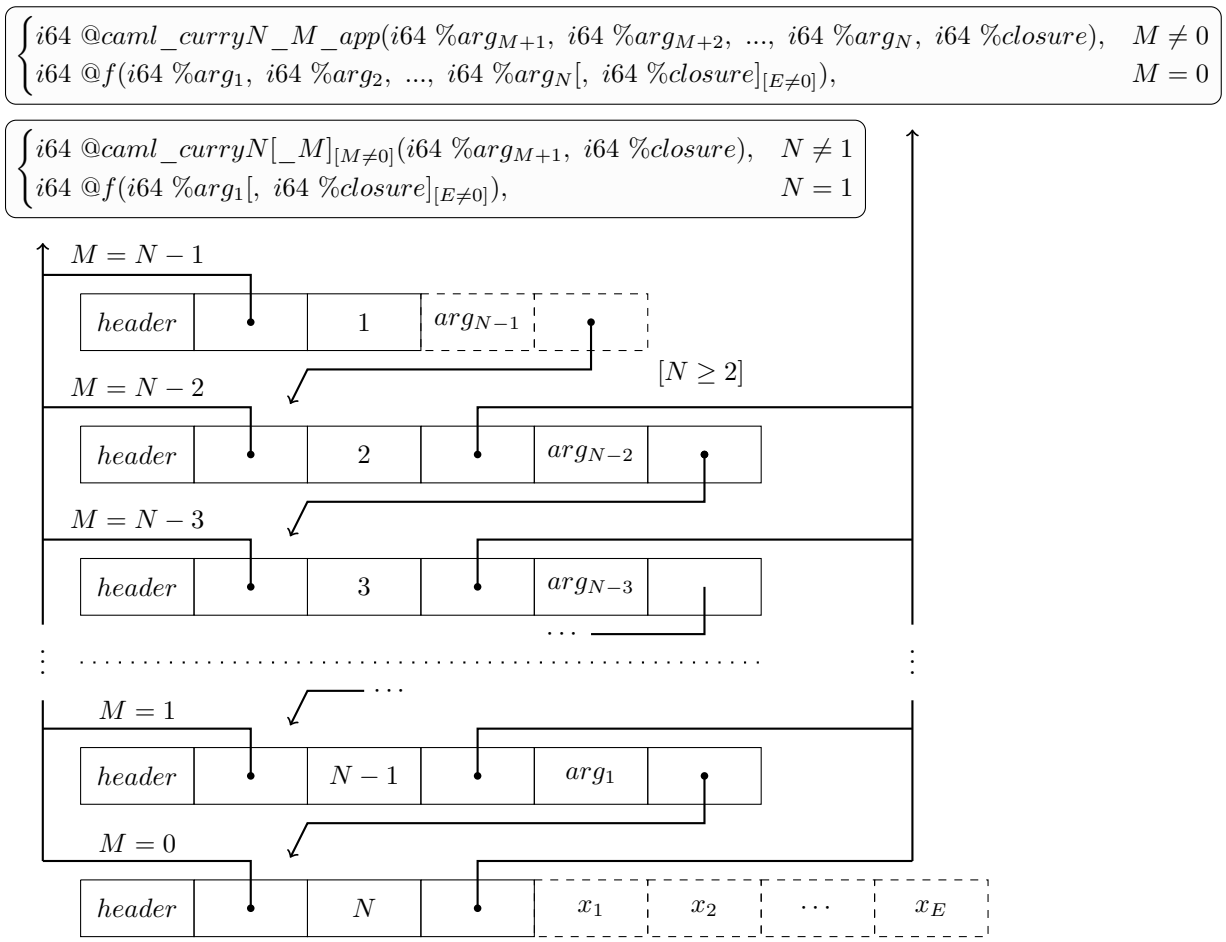

Рис. 1: Представление замыканий и частично применённых функций в OCaml

2.9.1 Нелокальные переходы setjmp/longjmp

Нелокальный переход — это восстановление состояния машины, ранее сохраненного в специальном буфере jmp_buf. Сохраняемое состояние включает адреса следующей исполняемой инструкции и вершины стека вызовов, потому может рассматриваться как безусловный переход, который способен пересекать границы функций. Для реализации исключений с помощью таких переходов во время выполнения программы поддерживается глобально доступный односвязный список вложенных защищённых блоков кода (try-блоков). Каждый узел такого списка располагается в записи активации одной из функций и содержит в качестве полезной нагрузки проинициализированный буфер jmp_buf. При возникновении исключения выполняется нелокальный переход с использованием буфера в начале списка и поиск обработчика возникшего исключения в соответствующем защищённом блоке. Такой поиск в OCaml является сопоставлением с образцом. Если обработчик найден, управление передаётся ему, иначе выполняется нелокальный переход с использованием следующего буфера из глобального списка. Для поддержания этого списка для каждого защищённого блока в теле функции в её записи активации резервируется память для узла списка. При каждой передаче управления в защищённый блок его буфер jmp_buf инициализируется и добавляется в начало списка. Если защищённый блок завершается без возникновения исключительных ситуаций, то первый элемент этого списка

удаляется из него.

Размер буфера jmp_buf около 200 байт, потому один из недостатков этого метода — увеличение потребления памяти. Также такой подход создаёт накладные расходы (на инициализацию буфера), даже если исключительные ситуации не возникают.

2.9.2 Табличный метод Intel Itanium C++ ABI

Единственный рекомендованный разработчиками LLVM способ реализации исключений — табличный метод, аналогичный используемому в компиляторах C++ проектов GCC и Clang. Он опирается на генерируемую компиляторным бэкендом информацию о записях активаций функций (Call Frame Information[15]) в формате DWARF-2 [16], которая позволяет определять границы записей активации и выполнять раскрутку стека вызовов (stack unwinding), выяснять, содержит ли текущая запись обработчик возникшей исключительной ситуации и в случае положительного ответа выполнять передачу управления в нужную точку программы с восстановлением состояния машины (машинных регистров), в котором она находилась на момент передачи управления в защищённый блок. Каждая такая точка называется посадочной площадкой (landing pad). Часть метода, не зависящая от языка, представлена библиотекой unwind. С её помощью можно генерировать исключения, во время их обработки изменять значения машинных регистров, получать доступ к сгенерированной компилятором отладочной информации, выполнять другие действия. При генерации исключительной ситуации библиотека unwind выполняет два прохода по стеку вызовов, первый используется для поиска посадочной площадки, содержащей обработчик возникшего исключения, второй — для выполнения различных действий, которые должны предшествовать передаче управления обработчику, например, в C++ для вызова деструкторов объектов. Содержит ли текущая площадка обработчик должна заключить специальная, специфичная для языка и его реализации функция — personality routine. Она же должна выполнять корректировку значений регистров (например, для передачи обработчику экземпляра возникшего исключения), вызовы деструкторов и т.п. Поддержка этого механизма в LLVM осуществлена специфичным для C++ образом. Так, выбор обработчика исключения в C++ осуществляется по его типу. Тип может быть легко представлен компактным образом — в форме указателя на экземпляр std::type_info, являющийся частью поддержки информации о типах времени выполнения (RTTI) в C++. LLVM предоставляет способ назначить базовый блок посадочной площадкой и описать, исключения каких именно типов он способен обрабатывать в терминах селекторов, которыми в случае C++ являются указатели на экземпляры std::type_info. Эта информация доступна personality routine, что позволяет механизму обработки исключений выполнять поиск подходящей площадки без повторной генерации исключения. В OCaml поиск обработчика является сопоставлением с образцом, которое может включать сравнение на равенство больших древовидных структур или числовых значений. Никакого аналога RTTI C++ в OCaml нет. По этой причине механизм селекторов, предоставляемый LLVM, был проигно-

[15] см. также https://sourceware.org/binutils/docs-2.24/as/CFI-directives.html#CFI-directives

рирован. Реализованная в рамках библиотеки поддержки времени исполнения personality routine всегда успешно завершает поиск обработчика в текущей посадочной площадке и передаёт ей управление. В пределах посадочной площадки выполняется сопоставление с образцом, и если оно завершится неудачей, выполняется повторная генерация исключения. Для чтения отладочной информации была использована библиотека поддержки времени исполнения[16] C++ из проекта Clang.

В LLVM IR вызов функции, которая способна сгененировать исключение, необходимо осуществлять специальным образом: с указанием посадочной площадки, которой должно быть передано управление в случае, если возникнет исключение. Для генерации соответствующего LLVM IR используется приём, практически идентичный описанному в 2.9.1, с той разницей, что поддерживаемый список содержит ссылки на посадочные площадки и существует только во время компиляции.

К достоинствам описанного метода относят отсутствие накладных расходов в случае, если исключительные ситуации не возникают. Однако перехват исключения в стиле Intel Itanium C++ ABI выполняется более, чем на два порядка медленнее, чем оригинальным механизмом обработки исключений OCaml. В частности, ввиду отсутствия в OCaml, как и в большинстве языков со сборкой мусора, деструкторов объектов, двойной проход по стеку вызовов представляется избыточным.

2.10 Поддержка сборки мусора

Встроенная в LLVM поддержка сборки мусора требует все значения, которые могут быть элементами корневого множества, регистрировать особым образом: резервировать для них в пределах первого базового блока память в записи активации функции и вызывать встроенную функцию llvm.gc.root с указателем на такую память в качестве аргумента. Так как практически любое значение OCaml может быть указателем на блок в куче, то есть, элементом корневого множества, подобная регистрация осуществляется для любого значения, полученного в качестве результата либо вызова функции, либо чтения значения из памяти по указателю. Более того, ввиду SSA-формы такую регистрацию приходится проводить не для каждой переменной, способной хранить элемент корневого множества, а для каждого возможного значения такой переменной. Эти особенности приводят к тому, что корневое множество содержит большое количество дубликатов, значений, не являющихся указателями и нулевых значений (последние появляются в том числе благодаря анализу времени жизни объектов, который является одной из оптимизаций, выполняемых LLVM). Это существенно увеличивает в размерах записи активаций функций и время работы сборщика мусора. Манипуляции со значениями, объявленными элементами корневого множества, практически не оптимизируются LLVM, в частности, такие значения никогда не размещаются в машинных регистрах.

[16] http://libcxx.llvm.org/

3 Результаты

В результате проделанной работы был реализован транслятор из одного из промежуточных представлений кода ocamlopt в LLVM IR и библиотека поддержки времени исполнения, которые вместе с оптимизатором и компилятором проекта LLVM и начальными стадиями оригинального компилятора ocamlopt составляют оптимизирующий компилятор с некоторого, достаточно полного, подмножества языка OCaml в машинный код для платформы Linux x86-64. В частности разработанный компилятор поддерживает:

- целочисленную и вещественную арифметику;
- основные типы данных, структуры и кортежи;
- все средства манипулирования потоком управления, в том числе сопоставление с образцом;
- вызов функций, реализованных как на OCaml, так и на C;
- генерацию и обработку исключений двумя различными способами: с помощью механизма нелокальных переходов setjmp/longjmp и табличным методом в соответствии со спецификацией Intel Itanium C++ ABI;
- замыкания и частично применённые функции;
- некоторые функции ввода-вывода;
- массивы и строки, в том числе модуль Array стандартной библиотеки OCaml.

Не была реализована поддержка многофайловой компиляции, объектно-ориентированного расширения OCaml, хеширования (и зависящих от него конструкций языка, например, полиморфных вариантов).

Ключевым ограничением полученной реализации является отсутствие сборщика мусора, хотя вся необходимая ему поддержка со стороны компилятора осуществляется, в том числе управление корневым множеством.

Тестирование корректности реализации осуществляется автоматизированными тестами, число которых на текущий момент — 67.

По сравнению с подходом, использованным в [2], подход, реализованный в рамках данной курсовой работы, снимает зависимость реализации от существующего в OCaml сборщика мусора и позволяет разработать подходящий самостоятельно, осуществляя со стороны LLVM IR необходимую поддержку, хотя реализация собственного сборщика мусора не вошла в работу. Также в отличие от упомянутой работы удалось провести несколько простых тестов производительности, некоторые из которых (записанные в процедурном стиле и выполняющие преимущественно арифметические расчёты) показали двухкратный прирост производительности по отношению к компилятору ocamlopt, хотя исчерпывающим проведённое тестирование ни в коем случае назвать нельзя. Последнее проведено не

было ввиду неполной поддержки возможностей языка, в частности, сборки мусора. Ещё одно отличие заключается в том, что в данной работе был реализован табличный механизм обработки исключений. Данная курсовая работа является единственной работой (среди рассмотренных), использующей для генерации LLVM IR функции из LLVM API для OCaml.

Репозиторий проекта доступен по адресу: https://github.com/ramntry/ocamlnc/.

Заключение

В результате проделанной работы (в том числе обзора связанных работ) удалось выявить ряд недостатков LLVM, часть которых, возможно, являются специфичными для языка OCaml. Обнаруженные недостатки можно разделить на несколько групп.

1. Сборка мусора. На текущий момент следует признать сомнительной возможность реализации перемещающего сборщика мусора при поддержке LLVM ввиду отсутствия в этой инфраструктуре штатных механизмов изменения указателей на объекты по инициативе сборщика мусора и небезопасности подобных манипуляций. Этот факт существенно препятствует реализации автоматического управления памятью, не страдающего фрагментацией памяти и обеспечивающего быстрое её выделение. Так как функциональные и объектно-ориентированные языки склонны к частым выделениям памяти небольшими блоками, это свойство LLVM особенно существенно для них. Также нарекания может вызывать механизм управления корневым множеством, который вместе с особенностями SSA-формы LLVM IR приводит к многократному размножению его элементов, что приводит к существенному увеличению среднего размера записи активации функции и увеличению времени, необходимого на анализ корневых указателей. Решить эти проблемы возможно отказом от использования встроенной в LLVM поддержки сборки мусора.

2. Исключения. LLVM IR имеет ограниченный набор средств манипулирования потоком управления и полностью абстрагирует вызов функций и структуру их записей активации. В связи с этим реализация исключений возможна только штатными способами. В LLVM существует частичная поддержка механизма setjmp/longjmp и полная поддержка механизма, описанного в спецификации Intel Itanium C++ ABI. Первый создаёт ненулевые накладные расходы как при простом входе в защищённый блок, так и при перехвате исключения и требует выделения не меньше 200 байт в записи активации функции на каждый защищённый блок. Второй относительно бесплатен при условии, что исключительных ситуаций не возникает, но крайне дорог, если это всё-таки происходит. Таким образом, перенос языков, полагающихся на эффективную реализацию исключений, в инфраструктуру LLVM может привести к существенному снижению производительности. OCaml относится к таким языкам.

3. Дизайн LLVM IR. Строго типизированную SSA-форму следует признать неудачным промежуточным представлением для полиморфного кода ввиду неосуществимости статической типизации значений, тип которых определяется динамически. Одновременно оказываются бессмысленными любые оптимизации кода, рассчитывающие на качественную типизацию промежуточного представления.

В целом инфраструктура LLVM проявила существенно большую зависимость от входного языка и тяготение к языкам семейства C/C++, чем ожидалось в начале этой работы, и особенно слабую подготовленность к языкам с богатой и эффективной поддержкой времени исполнения и выраженным полиморфизмом.

Исходя из опыта предшествующих работ можно предположить, что эффективная реализация как сборки мусора, так и механизма обработки исключений для OCaml воз-

можны, но требуют расширения LLVM (в первую очередь специфичными соглашениями о вызовах) и значительных временных затрат.

В ходе работы был изучен язык OCaml и закреплены навыки разработки на функциональных языках, осуществлено знакомство с инфраструктурой LLVM, приобретён опыт и знания в области дизайна и реализации компиляторов, в особенности в части промежуточных представлений кода и способов поддержки нетривиальных конструкций языка: замыканий и исключений.

Список литературы

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: principles, techniques, and tools. 2nd ed. Addison Wesley. 2007.

[2] Benner C. An LLVM Backend for OCaml // OCaml Users and Developers Workshop. Copenhagen, Denmark. September 2012. http://oud.ocaml.org/2012/.

[3] David A. Terei, Manuel M. T. Chakravarty. An LLVM Backend for GHC. University of New South Wales. 2010.

[4] International Standard ISO/IEC 9899:201x. Programming languages — C. 2011.

[5] Sagonas K., Stavrakakis C., Tsiouris Y. ErLLVM: An LLVM Backend for Erlang. 2012.

[6] Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. University of Illinois at Urbana-Champaign. 2004.

[7] Lattner C., Adve V. LLVM Language Reference Manual // http://llvm.org/releases/3.3/docs/LangRef.html

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler // IEEE Computer, December 1996.

[9] Norman Ramsey, Simon Peyton Jones, Christian Lindig. The C−− Language Specification Version 2.0 (CVS Revision 1.128). February 23, 2005 // http://www.cs.tufts.edu/~nr/c--/.

[10] Richard M. Stallman and the GCC Developer Community. GNU Compiler Collection Internals. For gcc version 4.10.0 (pre-release). 2014 // http://gcc.gnu.org/onlinedocs/.

[11] Simon Peyton Jones, Thomas Nordin, Dino Oliva. C−−: A Portable Assembly Language // Workshop on Implementing Functional Languages. 1998.

[12] Steven S. Muchnick. Advanced Compiler Design and Implementation. USA: Academic Press. 1997.

[13] William von Hagen. The Definitive Guide to GCC, Second Edition. Apress. 2006.

[14] Xavier Leroy and others. The OCaml system. Release 4.01. Documentation and user's manual. INRIA. September 12, 2013.

[15] Yaron Minsky, Anil Madhavapeddy, Jason Hickey. Real World OCaml. Functional programming for the masses. O'Reilly Media. November, 2013.

[16] UNIX International Programming Languages SIG. DWARF Debugging Information Format. Revision: 2.0.0. 1993 // http://www.dwarfstd.org/.