



LLVM: инфраструктура для разработки компиляторов

Бруно Кардос Лопес

Рафаэль Аулер

[РАСКТ]
PUBLISHING

ОМК
ИЗДАТЕЛЬСТВО

Бруно Кардос Лопес
Рафаэль Аулер

LLVM: инфраструктура для разработки компиляторов

Знакомство с основами LLVM
и использование базовых библиотек
для создания продвинутых инструментов

Getting Started with LLVM Core Libraries

Get to grips with LLVM essentials and use the core libraries to build advanced tools

Bruno Cardoso Lopes

Rafael Auler

[PUBLISHING] open source
community experience distilled

BIRMINGHAM - MUMBAI

LLVM: инфраструктура для разработки компиляторов

Знакомство с основами LLVM
и использование базовых библиотек
для создания продвинутых инструментов



Москва, 2015

УДК 004.4'422LLVM

ББК 32.973.33

Л77

Л77 Бруно Кардос Лопес, Рафаэль Аулер

LLVM: инфраструктура для разработки компиляторов. / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 342 с.: ил.

ISBN 978-5-97060-305-5

LLVM – новейший фреймворк для разработки компиляторов. Благодаря простоте расширения и организации в виде множества библиотек, LLVM легко поддается освоению даже начинающими программистами, вопреки устоявшемуся мнению о сложности разработки компиляторов.

Сначала эта книга покажет, как настроить, собрать и установить библиотеки, инструменты и внешние проекты LLVM. Затем познакомит с архитектурой LLVM и особенностями работы всех компонентов компилятора: анализатора исходных текстов, генератора кода промежуточного представления, генератора выполняемого кода, механизма JIT-компиляции, возможностями кросс-компиляции и интерфейсом расширений. На множестве наглядных примеров и фрагментов исходного кода книга поможет вам войти в мир разработки компиляторов на основе LLVM.

Издание предназначено энтузиастам, студентам, а также разработчикам компиляторов, интересующимся LLVM. Читатели должны знать язык программирования C++ и, желательно, иметь некоторые представления о теории компиляции.

Original English language edition published by Published by Packt Publishing Ltd., Livery Place, 35 Livery Street, Birmingham B3 2PB, UK. Copyright © 2014 Packt Publishing. Russian-language edition copyright (c) 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78216-692-4 (англ.)

ISBN 978-5-97060-305-5 (рус.)

Copyright © 2014 Packt Publishing

© Оформление, перевод на русский язык,
ДМК Пресс, 2015

ОГЛАВЛЕНИЕ

Об авторах	11
О рецензентах.....	12
Предисловие	14
Содержание книги.....	17
Что потребуется для работы с книгой.....	19
Кому адресована эта книга.....	19
Типографские соглашения	20
Отзывы и пожелания	21
Скачивание исходного кода примеров	21
Список опечаток.....	22
Нарушение авторских прав	22
Глава 1. Сборка и установка LLVM	23
Порядок нумерации версий LLVM.....	24
Установка скомпилированных пакетов LLVM	25
Установка скомпилированных пакетов с официального сайта.....	25
Установка с использованием диспетчера пакетов.....	27
Сборка из исходных текстов.....	28
Системные требования.....	28
Получение исходных текстов.....	29
Сборка и установка LLVM	30
Windows и Microsoft Visual Studio	37
Mac OS X и Xcode	41
В заключение	45
Глава 2. Внешние проекты.....	47
Введение в дополнительные инструменты Clang.....	47
Сборка и установка дополнительных инструментов Clang.....	49
Compiler-RT	50
Compiler-RT в действии.....	51
Расширение DragonEgg.....	52
Сборка DragonEgg	53

Конвейер компиляции с применением DragonEgg и инструментов LLVM	54
Пакет тестов LLVM	56
Использование LLDB.....	57
Введение в стандартную библиотеку libc++	59
В заключение	63
Глава 3. Инструменты и организация	64
Введение в основные принципы организации LLVM	64
LLVM сегодня	67
Взаимодействие с драйвером компилятора.....	71
Использование автономных инструментов.....	72
Внутренняя организация LLVM	75
Основные библиотеки LLVM	76
Приемы программирования на C++ в LLVM	78
Эффективные приемы программирования на C++ в LLVM.....	80
Демонстрация расширяемого интерфейса проходов.....	83
Реализация первого собственного проекта LLVM	84
Makefile.....	85
Реализация.....	87
Общие советы по навигации в исходных текстах LLVM	89
Читайте код как документацию	89
Обращайтесь за помощью к сообществу	90
Знакомьтесь с обновлениями – читайте журнал изменений в SVN как документацию	90
Заключительные замечания.....	93
В заключение	93
Глава 4. Анализатор исходного кода	94
Введение в Clang.....	94
Работа анализатора исходного кода	95
Библиотеки.....	97
Диагностика в Clang.....	100
Этапы работы анализатора Clang	105
Лексический анализ.....	105
Синтаксический анализ	112
Семантический анализ.....	119
Все вместе	122
В заключение	126
Глава 5. Промежуточное представление LLVM	127
Обзор.....	127
Зависимость LLVM IR от целевой архитектуры	130

Основные инструменты для работы с форматами IR	131
Введение в синтаксис языка LLVM IR	132
Представление LLVM IR в памяти	136
Реализация собственного генератора LLVM IR	139
Сборка и запуск генератора IR	143
Как генерировать любые конструкции IR с использованием генератора кода C++	144
Оптимизация на уровне IR	145
Оптимизации времени компиляции и времени компоновки	145
Определение проходов, имеющих значение	147
Зависимости между проходами	149
Прикладной интерфейс проходов	151
Реализация собственного прохода	152
В заключение	157
Глава 6. Генератор выполняемого кода	158
Обзор	158
Инструменты генераторов кода	161
Структура генератора кода	162
Библиотеки генераторов кода	163
Язык TableGen	165
Язык	167
Использование файлов .td с генераторами кода	168
Этап выбора инструкций	174
Класс SelectionDAG	175
Упрощение	178
Объединение DAG и легализация	179
Выбор инструкций с преобразованием «DAG-to-DAG»	181
Визуализация процесса выбора инструкций	184
Быстрый выбор инструкций	185
Планирование инструкций	186
Маршруты инструкций	186
Определение опасностей	188
Единицы планирования	188
Машинные инструкции	188
Распределение регистров	189
Объединение регистров	191
Замена виртуальных регистров	196
Архитектурно-зависимые обработчики	197
Пролог и эпилог	198
Индексы кадров стека	199
Инфраструктура машинного кода	199

Инструкции MC	200
Эмиссия кода	200
Реализация собственного прохода для генератора кода	203
В заключение	206
Глава 7. Динамический компилятор	208
Основы механизма динамической компиляции в LLVM	209
Введение в механизм выполнения	210
Управление памятью	212
Введение в инфраструктуру llvm::JIT	213
Запись блоков двоичного кода в память	213
JITMemoryManager	214
Механизмы вывода целевого кода	214
Информация о целевой архитектуре	215
Практика применения класса JIT	217
Введение в инфраструктуру llvm::MCJIT	222
Механизм MCJIT	223
Как MCJIT компилирует модули	224
Диспетчер памяти	227
Использование механизма MCJIT	228
Инструменты компиляции LLVM JIT	231
Инструмент lli	231
Инструмент llvm-rtld	232
Дополнительные ресурсы	233
В заключение	234
Глава 8. Кросс-платформенная компиляция	235
Сравнение GCC и LLVM	236
Триады определения целевой архитектуры	238
Подготовка инструментария	240
Стандартные библиотеки C и C++	240
Библиотеки времени выполнения	241
Ассемблер и компоновщик	242
Анализатор исходного кода Clang	242
Кросс-компиляция с аргументами командной строки Clang	244
Параметры драйвера, определяющие архитектуру	244
Зависимости	245
Кросс-компиляция	246
Изменение корневого каталога	248
Создание кросс-компилятора Clang	250
Параметры настройки	250
Сборка и установка кросс-компилятора на основе Clang	251
Альтернативные методы сборки	252

Тестирование	254
Одноплатные компьютеры	254
Симуляторы	255
Дополнительные ресурсы	255
В заключение	256

Глава 9. Статический анализатор Clang 257

Роль статического анализатора.....	258
Сравнение классического механизма предупреждений со статическим анализатором Clang	258
Возможности механизма символического выполнения.....	262
Тестирование статического анализатора	265
Использование драйвера и компилятора	265
Получение списка доступных средств проверки.....	266
Использование статического анализатора в Xcode IDE	268
Создание графических отчетов в формате HTML	269
Анализ больших проектов	269
Расширение статического анализатора Clang собственными средствами определения ошибок	275
Архитектура проекта	275
Разработка собственного средства проверки	277
Дополнительные ресурсы	287
В заключение	289

Глава 10. Инструменты Clang и фреймворк LibTooling 290

Создание базы данных команд компиляции	290
Clang-tidy	292
Проверка исходного кода с помощью Clang-tidy	293
Инструменты рефакторинга	294
Clang Modernizer	295
Clang Apply Replacements.....	296
ClangFormat	298
Modularize.....	300
Module Map Checker.....	308
PPTrace	309
Clang Query.....	311
Clang Check	313
Удаление вызовов <code>c_str()</code>	314
Создание собственного инструмента.....	314
Определение задачи – создание инструмента рефакторинга кода на C++.....	315
Определение структуры каталогов для исходного кода.....	315

Шаблонный код инструмента	317
Использование предикатов AST	321
Создание обработчиков	326
Тестирование нового инструмента рефакторинга	328
Дополнительные ресурсы	329
В заключение	329
Предметный указатель	331



ОБ АВТОРАХ

Бруно Кардос Лопес (Bruno Cardoso Lopes) получил степень доктора информационных технологий в университете города Капинас (Campinas), Бразилия. С 2007 года участвует в разработке LLVM, с нуля реализовал поддержку архитектуры MIPS и сопровождал ее в течение нескольких лет. Также занимался разработкой поддержки x86 AVX и ассемблера ARM. В настоящее время занимается исследованиями приемов сжатия кода и сужения системы команд (Instruction Set Architecture, ISA). В прошлом занимался разработкой драйверов для операционных систем Linux и FreeBSD.

Рафаэль Аулер (Rafael Auler) защитил кандидатскую диссертацию в университете города Капинас (Campinas), Бразилия. Имеет степени магистра информационных технологий и бакалавра в области конструирования вычислительных машин, полученные в том же университете. В дипломной работе на степень магистра реализовал экспериментальную версию инструмента, автоматически генерирующего генераторы выполняемого кода (backends) для поддержки архитектур в LLVM на основе файлов описаний. В настоящее время работает над докторской диссертацией и исследует приемы динамической двоичной трансляции, динамической (Just-in-Time) компиляции и компьютерные архитектуры. Также является лауреатом премии «Microsoft Research 2013 Graduate Research Fellowship Award».

О РЕЦЕНЗЕНТАХ

Эли Бендерски (Eli Bendersky) уже 15 лет профессионально занимается программированием, имеет богатый опыт системного программирования, включая компиляторы, компоновщики и отладчики. С начала 2012 года является одним из основных разработчиков проекта LLVM.

Логан Чен (Logan Chien) получил степень магистра информационных технологий в национальном университете Тайваня. Занимается исследованием архитектуры компиляторов, виртуальных машин и приемов оптимизации во время компиляции. Принимал участие в нескольких проектах по разработке открытого программного обеспечения, включая LLVM, Android и другие. Написал несколько исправлений к механизму обработки исключений с нулевыми накладными расходами для архитектуры ARM (zero-cost exception handling mechanism) и расширений для ассемблера ARM, интегрированного в LLVM. В 2012 проходил стажировку в Google как инженер-программист, во время которой занимался интеграцией LLVM в Android NDK.

Цзя Лю (Jia Liu) во время учебы в колледже начал заниматься разработкой программного обеспечения для GNU/Linux и уже после учебы участвовал в нескольких проектах по разработке открытого программного обеспечения. В настоящее время отвечает за все, что связано с программным обеспечением в China-DSP.

Интересуется технологиями компиляции и работает в этом направлении уже несколько лет. В свое свободное время все так же участвует в нескольких открытых проектах, таких как LLVM, QEMU и GCC/Binutils.

Был нанят китайским производителем микропроцессоров, компанией Glarun Technology, более широко известную, как China-DSP. China-DSP – это поставщик высокопроизводительных процессоров цифровой обработки сигналов (Digital Signal Processor, DSP). Ос-

нову бизнеса компании составляют: проектирование процессоров, разработка системного программного обеспечения и создание встраиваемых платформ параллельной обработки для электроэнергетики, связи, автомобилестроения, приборостроения и бытовой электроники.

Я хочу сказать спасибо моим родителям за то, что я появился на свет. Спасибо моей подруге, наставляющей меня на путь истинный. Спасибо моим коллегам за счастье работать с ними.

Джон Шакмейстер (John Szakmeister) получил степень магистра электротехники и электроники в университете Джонса Хопкинса (Балтимор, Мериленд, США) и является сооснователем Intelesys Corporation (www.intelesyscorp.com). Профессионально занимается программированием уже более 15 лет и обожает возиться с компьютерами, операционными системами, сложными алгоритмами и любыми встраиваемыми системами. Ярый сторонник Open Source и в свое свободное время принимает участие в развитии многих открытых проектов. В свободное от программирования время Джон упорно тренируется, стремясь к своему черному поясу по ниндзюцу, или читает техническую литературу.

Я хочу поблагодарить мою супругу Энн (Ann) и двух моих мальчиков, Мэттью (Matthew) и Эндрю (Andrew), за их терпение и понимание, когда я был занят рецензированием этой книги.

ПРЕДИСЛОВИЕ

Проект LLVM начинался со страсти к компиляторам единственного человека, Криса Латтнера (Chris Lattner). В событиях, последовавших за выходом первой версии LLVM, и дальнейшем подключении к проекту других разработчиков явно прослеживается сценарий развития, характерный для многих успешных открытых проектов: они появляются не в недрах крупных компаний, а благодаря простому человеческому интересу к той или иной теме. Например, первое ядро Linux появилось, благодаря интересу финского студента к операционным системам и его стремлению понять и увидеть на практике, как в действительности работают операционные системы.

И Linux, и LLVM развились до уровня первоклассного программного обеспечения, способного конкурировать с коммерческими аналогами, во многом благодаря вкладу большого числа программистов. И было бы несправедливо приписывать успех любого большого проекта одному человеку. Однако переход от студенческого проекта к сложному и надежному программному продукту в сообществе Open Source во многом зависит от такого важного фактора, как привлечение сторонних разработчиков, кому пришлось бы по душе тратить свое время на проект.

В школах присутствует атмосфера увлекательности, потому что образование предполагает изучение внутреннего порядка вещей. Для учащихся познание тайн устройства сложных механизмов является собой процесс перехода от ощущения полной загадочности к чувству победы и уверенного владения знаниями. В такой среде, в университете города Урбана-Шампейн штата Иллинойс (UIUC), зародился проект LLVM, который использовался как прототип для дальнейших исследований и как учебное пособие для курса лекций по устройству компиляторов, который вел Викрам Адве (Vikram Adve), руководитель дипломного проекта Латтнера. Студенты помогали в выявлении первых ошибок и дали начальный импульс в развитии LLVM, как простого в изучении программного проекта с ясной архитектурой.

Вопиющее несоответствие теории и практики программирования сбивают с толку многих студентов – будущих программистов. Про-

стая и ясная концепция в теории может включать в себя столько наслоений из деталей реализации, что программные проекты становятся просто невозможно охватить мысленно. Хорошо продуманная архитектура с мощными абстракциями помогает человеческому мозгу разобраться во всех наслоениях и на всех уровнях: от общего представления, как работает программа, до мельчайших подробностей ее реализации.

Это особенно верно для компиляторов. Студенты, испытывающие желание узнать, как работают компиляторы, часто сталкиваются с трудной задачей, когда дело доходит до изучения фактической реализации компилятора. До появления LLVM, одним из немногих компиляторов с открытым исходным кодом, доступных хакерам¹ и любопытным студентам для изучения практической реализации, был GCC.

Любой программный проект достаточно ясно отражает представления программистов, создавших его. Это можно наблюдать в абстракциях, используемых для отделения модулей друг от друга и представления данных в разных компонентах. Программисты могут иметь разные взгляды на одно и то же. Соответственно, давнишние и крупные проекты, такие как GCC (который развивается уже более 30 лет), часто являются комбинацией взглядов и представлений целых поколений программистов, что делает такие проекты чрезвычайно сложными в изучении для вступающих в них новых программистов и любознательных исследователей.

Проект LLVM привлекает не только опытных программистов – разработчиков компиляторов, – но также множество юных и любознательных умов, которые видят в нем более ясный и простой для изучения программный продукт, представляющий собой компилятор с огромным потенциалом. Это очевидно из большого числа научных трудов, выбравших LLVM за основу для исследований. Причина проста; в научном сообществе за практические аспекты реализации часто отвечают студенты, поэтому для научно-исследовательских проектов первостепенное значение имеет простота кодовой базы, чтобы студенты легко могли овладеть ею. Соблазненные новейшей архитектурой на основе языка C++ (вместо C, на котором написан GCC), модульной структурой (вместо монолитной в GCC) и концепциями, которые легко укладываются в современную теорию построения ком-

¹ Здесь слово «хакер» используется в его первоначальном смысле: «высококвалифицированный ИТ-специалист, человек, который понимает тонкости работы программ ЭВМ». – *Прим. перев.*

пиляторов, многие исследователи отметили, насколько просто освоить LLVM и использовать для реализации своих идей. Успех LLVM в академической среде определенно стал следствием уменьшением разрыва между теорией и практикой.

Проект LLVM привлекает не только академическое сообщество, как инструмент для исследований, но и промышленность, из-за гораздо более либеральной лицензии, по сравнению с лицензией GPL, на условиях которой распространяется GCC. В академических кругах, где вырос проект, ложкой дегтя в бочке меда для исследователей, пишущих программный код, является страх, что он будет использоваться только в единственном эксперименте, по окончании которого этот код останется только выбросить. Чтобы исключить возможность такой участи, Крис Латтнер, давший жизнь LLVM в своем дипломном проекте, решил лицензировать свою разработку на условиях University of Illinois/NCSA Open Source License, допускающих коммерческое и некоммерческое использование продукта при сохранении упоминания об авторских правах. Цель состояла в том, чтобы максимально упростить распространение LLVM, и эта цель была достигнута в полной мере. В 2012 году, проект LLVM был удостоен премии «ACM Software System Award», присуждаемой за разработку программных систем, сыгравших важную роль в науке.

Многие компании воспользовались проектом LLVM для удовлетворения своих нужд и вносят свой вклад в его развитие, расширяя спектр языков, поддерживаемых LLVM, а также аппаратных архитектур, для которых эти компиляторы могут генерировать код. На этом новом этапе развития проекта было обеспечено непревзойденное совершенство библиотек и инструментов, позволившее навсегда оставить в прошлом уровень экспериментального академического программного продукта и перейти на уровень надежной основы для коммерческих разработок. Вместе с этим изменилось и имя проекта: название «Low Level Virtual Machine» (низкоуровневая виртуальная машина) сменилось на его аббревиатуру LLVM.

Решение сменить название «Low Level Virtual Machine» на аббревиатуру LLVM отражает изменение целей проекта. Первоначально LLVM создавалась как дипломный проект, с целью служить основой для изучения приемов программной оптимизации. Заложенные в проект идеи первоначально были опубликованы на международном симпозиуме 2003 MICRO (International Symposium on Microarchitecture), в статье под заголовком «LLVA: A Low-level Virtual Instruction Set Architecture», где описывался набор инструкций, и на

симпозиуме 2004 CGO (International Symposium on Code Generation and Optimization), в статье под заглавием «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation».

Выйдя за академические рамки, LLVM превратился в удачно спроектированный компилятор с интересным свойством – сохранением на диск промежуточного представления программного кода. В коммерческих системах LLVM никогда не использовался в роли настоящей виртуальной машины, как, например, виртуальная машина Java (Java Virtual Machine, JVM), поэтому не имело смысла сохранять прежнее название Low Level Virtual Machine. С другой стороны, в наследство остались некоторые другие любопытные названия. Файл на диске, где хранится программа в промежуточном представлении LLVM, обычно называют «LLVM bitcode» (LLVM-биткод), в пику названию «Java bytecode» (Java-байткод), подчеркивая разницу в размерах промежуточного представления в LLVM и Java.

Работая над этой книгой, мы ставили перед собой две цели. Во-первых, поскольку проект LLVM сильно разросся, мы хотели познакомить вас с небольшими его частями, по одной составляющей за раз, чтобы максимально упростить изучение и одновременно дать почувствовать радость владения мощной библиотекой компиляторов. Во-вторых, мы хотели вдохнуть в вас дух свободного хакерства, чтобы вы не ограничивали себя рамками концепций, представленных здесь, и никогда не прекращали расширять свой кругозор.

Успешной работы!

Содержание книги

Глава 1, «Сборка и установка LLVM», покажет, как установить пакет Clang/LLVM в Linux, Windows и Mac OS, а также, как собрать LLVM в Visual Studio и Xcode. Здесь также будут обсуждаться разные версии дистрибутивов LLVM и вопросы выбора лучшей версии для ваших условий: скомпилированные файлы, пакеты дистрибутивов или исходные коды.

Глава 2, «Внешние проекты», познакомит с внешними проектами LLVM, распространяемыми в отдельных пакетах или хранящимися в других репозиториях, такими как дополнительные инструменты Clang и расширение DragonEgg GCC, отладчик LLVM Debugger (LLDB) и пакет тестов LLVM.

Глава 3, «Инструменты и организация», описывает организацию инструментов в проекте LLVM, демонстрирует примеры их исполь-

зования для получения промежуточного кода из исходных текстов. Также рассказывает, как действует драйвер компилятора, и, наконец, как написать очень простой инструмент для LLVM.

Глава 4, «Анализатор исходного кода», знакомит с анализатором исходного кода LLVM – проектом Clang. Проведет вас через все этапы работы анализатора на примерах создания небольших программ, использующих каждую часть интерфейса. Завершается примером небольшого драйвера компилятора, использующего библиотеки Clang.

Глава 5, «Промежуточное представление LLVM», разъясняет ключевой аспект архитектуры LLVM: промежуточное представление (intermediate representation). Показывает отличительные характеристики этого представления, знакомит с синтаксисом, структурой и приемами разработки инструментов, генерирующих промежуточное представление LLVM IR.

Глава 6, «Генератор выполняемого кода», знакомит с устройством генератора выполняемого кода (backend) LLVM, ответственного за преобразование промежуточного представления LLVM IR в машинный код. Эта глава проведет вас через все этапы работы генератора кода, знание которых поможет вам создать собственный генератор для LLVM. Завершается примером создания такого генератора.

Глава 7, «Динамический компилятор», разъясняет инфраструктуру динамической компиляции (Just-in-Time) в LLVM, которая позволяет генерировать и выполнять машинный код по мере необходимости. Эта технология особенно востребована в приложениях, где во время выполнения имеется только исходный программный код, таких как интерпретаторы JavaScript в браузерах. Эта глава проведет вас через все этапы использования библиотек при разработке собственного JIT-компилятора.

Глава 8, «Кросс-платформенная компиляция», проведет вас через все этапы создания программ для других аппаратных архитектур, таких как ARM, с использованием Clang/LLVM. Покажет, как правильно настроить окружение для компиляции программ, которые будут выполняться за пределами этого окружения.

Глава 9, «Статический анализатор Clang», описывает мощный инструмент поиска ошибок в исходном программном коде до запуска программ за счет анализа этого кода. Данная глава также покажет, как дополнить статический анализатор Clang своими собственными средствами контроля.



Глава 10, «Инструменты Clang и фреймворк LibTooling», знакомит с фреймворком LibTooling и некоторыми встроенными инструментами Clang, дающими возможность выполнить рефакторинг исходного кода и простые виды его анализа. Эта глава завершается примером создания собственного инструмента рефакторинга исходного кода на языке C++ с применением данной библиотеки.

Когда мы работали над этой книгой, версия LLVM 3.5 еще не вышла. Но, несмотря на то, что в центре внимания этой книги находится версия LLVM 3.4, мы планируем обновить примеры до версии LLVM 3.5 к третьей неделе сентября 2014 года, чтобы дать вам возможность опробовать их с более новой версией LLVM. Обновленные примеры будут доступны по адресу: https://www.packtpub.com/sites/default/files/downloads/6924OS_Appendix.pdf.

Что потребуется для работы с книгой

Для исследования LLVM можно использовать операционные системы UNIX, Mac OS X и Windows, при наличии в них современного компилятора C++. Исходный код LLVM очень требователен к выбору компилятора C++ – компилятор должен соответствовать новейшим стандартам. Это означает, что в Linux вы должны использовать версию GCC не ниже 4.8.1; в Mac OS X версию Xcode не ниже 5.1; и в Windows – как минимум Visual Studio 2012.

Несмотря на то, что здесь рассказывается, как собрать LLVM в Windows с помощью Visual Studio, эта книга не нацелена на данную платформу, потому что некоторые возможности LLVM на ней недоступны. Например, в Windows LLVM не поддерживает загружаемые модули, но мы покажем, как писать расширения в виде разделяемых библиотек. Поэтому, чтобы познакомиться с такими возможностями на практике, необходимо использовать Linux или Mac OS X.

Если вы не желаете заниматься сборкой LLVM самостоятельно, воспользуйтесь пакетом готовых двоичных файлов. Правда скомпилированные дистрибутивы доступны не для всех платформ.

Кому адресована эта книга

Эта книга адресована энтузиастам, студентам, изучающим информационные технологии, и разработчикам компиляторов, интересую-

щимся фреймворком LLVM. Читатели должны знать язык программирования C++ и, желательно, иметь некоторые представления о теории компиляции. И для начинающих, и для опытных специалистов эта книга послужит практическим введением в LLVM, не содержащим сложных сценариев. Если вас интересует данная технология, тогда эта книга определена для вас.

Типографские соглашения

В этой книге используется несколько разных стилей оформления текста, с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, адреса URL, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении:

«Предварительно скомпилированные пакеты для Windows распространяются в виде простого в использовании инсталлятора, который сам распакует дерево каталогов LLVM в папку Program Files.»

Блоки программного кода оформляются так:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int main() {
    uint64_t a = 0ULL, b = 0ULL;
    scanf ("%lld %lld", &a, &b);
    printf ("64-bit division is %lld\n", a / b);
    return EXIT_SUCCESS;
}
```

Когда нам потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, мы будем выделять его жирным шрифтом:

```
KEYWORD(float , KEYALL)
KEYWORD(goto , KEYALL)
KEYWORD(inline , KEYC99|KEYCXX|KEYGNU)
KEYWORD(int , KEYALL)
KEYWORD(return , KEYALL)
KEYWORD(short , KEYALL)
KEYWORD(while , KEYALL)
```

Ввод и вывод в командной строке будет оформляться так:

```
$ sudo mv clang+llvm-3.4-x86_64-linux-gnu-ubuntu-13.10 llvm-3.4
$ export PATH="$PATH:/usr/local/llvm-3.4/bin"
```

Новые термины и важные определения будут выделяться в обычном тексте жирным. Надписи, которые вы будете видеть на экране, например в меню или в диалогах, будут выделяться в тексте так: «Во время установки не забудьте отметить флажок **Add CMake to the system PATH for all users** (Добавить CMake в системную переменную PATH для всех пользователей)».

Примечание. Так будут оформляться предупреждения и важные примечания.

Совет. Так будут оформляться советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.



ГЛАВА 1.

Сборка и установка LLVM

Фреймворк LLVM доступен для разных версий **Unix** (GNU/Linux, FreeBSD, Mac OS X) и **Windows**. В этой главе мы расскажем, какие шаги необходимо выполнить, чтобы получить действующий фреймворк LLVM во всех этих системах. Для некоторых систем доступны предварительно скомпилированные пакеты LLVM и Clang, но вы можете сами выполнить сборку из исходных текстов.

Пользователи, только начинающие изучать LLVM, должны знать, что компиляторы на основе LLVM включают в себя библиотеки и инструменты из обоих пакетов – LLVM и Clang. Поэтому все инструкции в этой главе описывают сборку и установку обоих пакетов. На протяжении всей книги мы будем подразумевать версию LLVM 3.4. Однако, имейте в виду, что LLVM – это молодой проект и продолжает активно развиваться, поэтому в последующих версиях некоторые аспекты могут измениться.

Совет. Когда мы работали над этой книгой, версия LLVM 3.5 еще не вышла. Но, несмотря на то, что в центре внимания этой книги находится версия LLVM 3.4, мы планируем обновить примеры до версии LLVM 3.5 к третьей неделе сентября 2014 года, чтобы дать вам возможность опробовать их с более новой версией LLVM. Обновленные примеры будут доступны по адресу: https://www.packtpub.com/sites/default/files/downloads/69240S_Appendix.pdf.

В этой главе рассматриваются следующие темы:

- ♦ порядок нумерации версий LLVM;
- ♦ установка предварительно скомпилированных файлов LLVM;
- ♦ установка LLVM с использованием диспетчера пакетов;
- ♦ сборка LLVM из исходных текстов для Linux;
- ♦ сборка LLVM из исходных текстов для Windows и Visual Studio;
- ♦ сборка LLVM из исходных текстов для Mac OS X и Xcode.

Порядок нумерации версий LLVM

Проект LLVM продолжает быстро развиваться, благодаря усилиям многих программистов. За 10 лет после выпуска первой версии и до момента появления версии 3.4, в репозиторий SVN (в качестве системы управления версиями используется Subversion) было передано более 200 000 исправлений и изменений. За один только 2013 год в репозиторий было передано 30 000 новых изменений. Как следствие, постоянно появляются новые возможности, а прежние – быстро устаревают. Как и в любом другом крупном проекте, разработчики должны точно следовать плотному графику выпуска стабильных версий, уверенно проходящих разнообразные тесты, чтобы дать пользователям возможность испытать новые возможности.

На протяжении всей своей истории проект LLVM следует стратегии выпуска двух стабильных версий в год. В каждой следующей версии увеличивается младший номер. Например, при переходе от версии 3.3 к версии 3.4 изменяется младший номер версии. Когда младший номер достигает значения 9, в следующей версии будет увеличен старший номер, например, после версии LLVM 2.9 была выпущена версия LLVM 3.0. Изменение старшего номера версии вовсе не означает наличие существенных изменений, в сравнении с предыдущей версией, а всего лишь служит границей очередного пятилетнего отрезка развития проекта.

Для проектов, зависящих от LLVM, типично использовать *стволовую (trunk)* версию, то есть, наиболее свежую, доступную в репозитории SVN, которая иногда может оказаться нестабильной. Недавно, начиная с версии 3.4, в сообществе LLVM было решено выпускать *промежуточные версии*, вводя дополнительный номер версии – номер выпуска. Первым результатом такого решения стал выход версии LLVM 3.4.1. Цель выпуска промежуточных версий – включение исправлений из стволовой версии в последнюю помеченную (*tagged*) версию, не имеющую новых особенностей, чтобы обеспечить полную совместимость. Промежуточные версии должны выходить через три месяца после выхода последней основной версии. Так как такая нумерация версий еще достаточно нова, мы сосредоточимся на установке версии LLVM 3.4. Существует большое число предварительно скомпилированных пакетов LLVM 3.4, но, следуя нашим инструкциям, вы без особых усилий сможете самостоятельно собрать LLVM 3.4.1 или более новую версию.

Установка скомпилированных пакетов LLVM

Чтобы упростить задачу установки программного обеспечения и избавить вас от необходимости выполнять компиляцию самостоятельно, поставщики LLVM создают предварительно скомпилированные пакеты для определенных платформ. Компиляция программного обеспечения иногда может оказаться очень непростым делом; она может потребовать некоторого времени и должна выполняться, только если вы используете другую платформу или принимаете активное участие в разработке проекта. Поэтому, для желающих побыстрее приступить к исследованию LLVM, существуют предварительно скомпилированные пакеты. Однако, в этой книге мы советуем использовать исходные тексты LLVM. Вы должны быть готовы скомпилировать LLVM самостоятельно.

Существует два основных способа установки предварительно скомпилированных пакетов LLVM: на официальном веб-сайте или на сайтах проектов, выпускающих дистрибутивы для GNU/Linux и Windows.

Установка скомпилированных пакетов с официального сайта

На официальном сайте проекта LLVM можно загрузить следующие предварительно скомпилированные пакеты для версии 3.4:

Таблица 1.1. Официальные скомпилированные пакеты для версии LLVM 3.4

Архитектура	Версия
x86_64	Ubuntu (12.04, 13.10), Fedora 19, Fedora 20, FreeBSD 9.2, Mac OS X 10.9, Windows и openSUSE 13.1.
i386	openSUSE 13.1, FreeBSD 9.2, Fedora 19, Fedora 20 и openSUSE 13.1
ARMv7/ARMv7a	Linux

Более полный перечень вы найдете на сайте <http://www.llvm.org/releases/download.html>, в разделе **Pre-built Binaries** (предварительно скомпилированные пакеты) для версии LLVM, которую вы желали бы загрузить. Например, чтобы загрузить и установить LLVM в Ubuntu 13.10, можно выполнить следующие команды:

```
$ sudo mkdir -p /usr/local; cd /usr/local
$ sudo wget http://llvm.org/releases/3.4/clang+llvm-3.4-x86_64-linux-
gnu-ubuntu-13.10.tar.xz
$ sudo tar xvf clang+llvm-3.4-x86_64-linux-gnu-ubuntu-13.10.tar.xz
$ sudo mv clang+llvm-3.4-x86_64-linux-gnu-ubuntu-13.10 llvm-3.4
$ export PATH="$PATH:/usr/local/llvm-3.4/bin"
```

После этого LLVM и Clang готовы к использованию. Не забывайте о необходимости постоянно изменять системную переменную окружения `PATH`, потому что изменение, которое было выполнено в последней строке, действует только до завершения текущего сеанса. Чтобы проверить корректность установки, можно попробовать выполнить простую команду, которая выведет версию Clang:

```
$ clang -v
```

Если при попытке выполнить эту команду возникли какие-либо проблемы, попробуйте запустить выполняемый файл непосредственно из каталога, куда он был установлен, чтобы убедиться, что проблема не вызвана неправильной настройкой переменной окружения `PATH`. Если и в этом случае запустить Clang не удалось, возможно вы установили предварительно скомпилированную версию, несовместимую с вашей системой. Не забывайте, что во время компиляции, двоичные файлы компоуются с динамическими библиотеками определенных версий. Сообщение об отсутствии библиотек, появляющееся при попытке запустить приложение, служит явным признаком, что двоичные файлы были скомпилированы в системе, несовместимой с вашей.

Совет. В Linux, например, текст сообщения об отсутствии библиотеки содержит имя выполняемого файла и имя динамической библиотеки, которую не удалось загрузить. Обратите внимание на имя динамической библиотеки – это явный признак, что динамический компоновщик и загрузчик не смог загрузить данную библиотеку, потому что программа была собрана в несовместимой системе.

Чтобы установить предварительно скомпилированные пакеты в других системах (кроме Windows), можно выполнить точно такую же последовательность шагов. Предварительно скомпилированные пакеты для Windows распространяются в виде простого в использовании инсталлятора, который сам распакует дерево каталогов пакета LLVM в папку Program Files. Инсталлятор предусматривает возможность автоматического изменения переменной окружения `PATH`, чтобы можно было запускать Clang из любого окна командной строки.

Установка с использованием диспетчера пакетов

Диспетчеры пакетов доступны в самых разных системах и также упрощают получение и установку предварительно скомпилированных пакетов LLVM/Clang. Это наиболее предпочтительный путь для большинства пользователей, поскольку диспетчер пакетов автоматически разрешает зависимости и проверяет совместимость вашей системы с устанавливаемыми двоичными файлами.

Например, в Ubuntu (10.04 и выше) установку можно выполнить следующей командой:

```
$ sudo apt-get install llvm clang
```

В Fedora 18 установка выполняется так же, но используется иной диспетчер пакетов:

```
$ sudo yum install llvm clang
```

Обновление с использованием промежуточных сборок

Пакеты могут собираться по ночам, из самых свежих исходных текстов, содержащих самые последние изменения. Такие пакеты могут оказаться полезными для разработчиков и пользователей LLVM, для кого представляют интерес самые свежие версии, или сторонним пользователям, стремящимися обеспечить актуальность своих локальных проектов.

Linux

Для Debian и Ubuntu Linux (i386 и amd64) доступны ежедневные сборки, полученные компиляцией исходных текстов из репозитория LLVM. Подробности см. на сайте <http://llvm.org/apt>.

Например, установить такую сборку LLVM и Clang в Ubuntu 13.10 можно следующей последовательностью команд:

```
$ sudo echo "deb http://llvm.org/apt/raring/ llvm-toolchain-raring main"
>> /etc/apt/sources.list
```

```
$ wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add -
```

```
$ sudo apt-get update
```

```
$ sudo apt-get install clang-3.5 llvm-3.5
```

Windows

Инсталляторы ежедневных сборок LLVM/Clang для Windows доступны для загрузки по адресу: <http://llvm.org/builds/>, в разделе **Windows snapshot builds** (ежедневные сборки для Windows). По умолчанию инструменты LLVM/Clang устанавливаются в папку `C:\Program Files\LLVM\bin` (это местоположение может измениться, в зависимости от версии). Обратите внимание, что существует отдельный драйвер Clang `clang-cl.exe`, имитирующий `cl.exe` из Visual C++. Если вы предполагаете использовать классический драйвер, совместимый с GCC, используйте `clang.exe`.

Совет. *Имейте в виду, что ежедневные сборки не являются стабильными версиями и могут работать очень неустойчиво.*

Сборка из исходных текстов

В отсутствие предварительно скомпилированных пакетов, LLVM и Clang можно скомпилировать из исходных текстов. Этот путь поможет поближе познакомиться со структурой LLVM. Кроме того, вы сможете точнее настроить некоторые параметры сборки.

Системные требования

По адресу: <http://llvm.org/docs/GettingStarted.html#hardware> можно найти постоянно обновляемый список платформ, поддерживаемых проектом LLVM. Полный перечень предварительных требований, которые должны быть соблюдены перед компиляцией LLVM, можно найти по адресу: <http://llvm.org/docs/GettingStarted.html#software>. В системах Ubuntu, например, разрешить зависимости можно с помощью команды:

```
$ sudo apt-get install build-essential zlib1g-dev python
```

Если вы пользуетесь довольно старой версией дистрибутива Linux с устаревшими пакетами, найдите время, чтобы обновить свою систему. Исходные тексты LLVM написаны на языке C++ и очень требовательны к версии компилятора, с помощью которого они будут компилироваться, поэтому попытка скомпилировать их старой версией компилятора C++ с большой долей вероятности не увенчается успехом.

Получение исходных текстов

Исходные тексты LLVM распространяются на условиях BSD-подобной лицензии и могут быть загружены с официального сайта проекта или из репозитория SVN. Чтобы получить исходные тексты для версии 3.4, можно перейти по адресу <http://llvm.org/releases/download.html#3.4> или напрямую загрузить их, как показано ниже. Обратите внимание, что вам всегда нужны будут исходные тексты Clang и LLVM, а исходные тексты дополнительных инструментов Clang (clang-tools-extra) можно загружать и устанавливать по желанию. Однако эти инструменты потребуются тем, кто пожелает опробовать примеры из главы 10, «Инструменты Clang и фреймворк LibTooling». Информация по сборке дополнительных проектов приводится в следующей главе. Используйте следующие команды для загрузки и установки LLVM, Clang и дополнительных инструментов Clang:

```
$ wget http://llvm.org/releases/3.4/llvm-3.4.src.tar.gz
$ wget http://llvm.org/releases/3.4/clang-3.4.src.tar.gz
$ wget http://llvm.org/releases/3.4/clang-tools-extra-3.4.src.tar.gz
$ tar xzf llvm-3.4.src.tar.gz; tar xzf clang-3.4.src.tar.gz
$ tar xzf clang-tools-extra-3.4.src.tar.gz
$ mv llvm-3.4 llvm
$ mv clang-3.4 llvm/tools/clang
$ mv clang-tools-extra-3.4 llvm/tools/clang/tools/extra
```

Примечание. *Загруженные исходные тексты в Windows можно распаковать с помощью gunzip, WinZip или любого другого архиватора.*

SVN

Чтобы получить исходные тексты непосредственно из репозитория SVN, убедитесь сначала, что в вашей системе установлен пакет subversion. На следующем шаге решите: хотите ли вы получить последнюю версию, хранящуюся в репозитории, или стабильную. Чтобы получить последнюю (стволовую) версию, выполните следующую последовательность команд (предполагается, что они запускаются из каталога, подготовленного для размещения исходных текстов):

```
$ svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
$ cd ../projects
```

```
$ svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
$ cd ../tools/clang/tools
$ svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

Чтобы получить стабильную версию (например, версию 3.4), замените слово `trunk` на `tags/RELEASE_34/final` во всех командах. Вас может также заинтересовать возможность навигации по репозиторию LLVM SVN, с целью посмотреть историю включения в него изменений, журналы и структуру дерева каталогов с исходными текстами. Для этого откройте страницу <http://llvm.org/viewvc>.

Git

Существует также возможность получить исходные тексты из Git-зеркала репозитория, синхронизированного с репозиторием SVN:

```
$ git clone http://llvm.org/git/llvm.git
$ cd llvm/tools
$ git clone http://llvm.org/git/clang.git
$ cd ../projects
$ git clone http://llvm.org/git/compiler-rt.git
$ cd ../tools/clang/tools
$ git clone http://llvm.org/git/clang-tools-extra.git
```

Сборка и установка LLVM

В этом разделе описываются разные способы сборки и установки LLVM.

С использованием сценария `configure`, сгенерированного с помощью `autotools`

Стандартный способ сборки LLVM заключается в том, чтобы создать платформу-зависимые файлы сборки с применением сценария `configure`, сгенерированного с помощью системы сборки GNU `autotools`. Эта система сборки пользуется большой популярностью и многие из вас наверняка знакомы с ней. Она поддерживает разные параметры настройки.

Примечание. Устанавливать систему сборки GNU `autotools` необходимо, только если вы намереваетесь подменить систему сборки LLVM и сгенерировать новый сценарий `configure`. Обычно в этом нет никакой необходимости.

Потратьте некоторое время, чтобы познакомиться с имеющимися параметрами настройки, используя следующие команды:

```
$ cd llvm
$ ./configure --help
```

Ниже приводится описание некоторых из них:

- `--enable-optimized`: Этот параметр позволяет скомпилировать LLVM/Clang без отладочной информации и с оптимизациями. По умолчанию выключен. Включение отладочной информации и запрет оптимизаций рекомендуется, когда библиотеки LLVM используются для разработки, но перед выпуском готовой версии оптимизацию следует включить, потому что отсутствие оптимизации существенно замедляет работу LLVM.
- `--enable-assertions`: Этот параметр включает дополнительные проверки в коде и очень полезен для разработки основных библиотек LLVM. По умолчанию он включен.
- `--enable-shared`: Этот параметр позволяет скомпилировать библиотеки LLVM/Clang в виде разделяемых библиотек и скомпоновать инструменты LLVM с ними. Если вы планируете разрабатывать инструменты за пределами системы сборки LLVM и желаете динамически компоновать их с библиотеками LLVM, включите этот параметр. По умолчанию он выключен.
- `--enable-jit`: Этот параметр включает **динамическую компиляцию** (Just-In-Time Compilation) для всех целей, поддерживающих такую возможность. По умолчанию он включен.
- `--prefix`: Определяет путь к каталогу установки, куда будут сохраняться скомпилированные файлы инструментов и библиотек LLVM/Clang при установке; например, если определить параметр `--prefix=/usr/local/llvm`, выполняемые файлы будут устанавливаться в каталог `/usr/local/llvm/bin`, а библиотеки – в каталог `/usr/local/llvm/lib`.
- `--enable-targets`: Этот параметр позволяет выбрать множество целей, для компиляции. Стоит напомнить, что с помощью LLVM можно выполнять кросс-компиляцию, то есть, компилировать программы, которые должны работать на других платформах, таких как ARM, MIPS и т. д.. Данный параметр определяет, какие генераторы выполняемого кода (backends) должны быть включены в библиотеки, отвечающие за созда-

ние машинного кода. По умолчанию компилируются все цели, но вы можете сократить время компиляции, определив только те цели, которые вас интересуют.

Совет. *Этого параметра недостаточно, чтобы получить автономный кросс-компилятор. За дополнительной информацией по этой теме обращайтесь к главе 8, «Кросс-платформенная компиляция».*

После запуска сценария `configure` с желаемыми параметрами, сборку можно завершить классической парой команд `make` и `make install`. А теперь перейдем к примеру.

Сборка и установка в Unix

В этом примере мы скомпилируем неоптимизированную версию LLVM/Clang (с отладочной информацией), выполнив последовательность команд, которые присутствуют в любой Unix-подобной системе или в оболочке Cygwin. Вместо каталога установки `/usr/local/llvm`, как было показано в предыдущих примерах, компиляция и установка будут выполняться в домашний каталог, что позволяет установить LLVM, не имея привилегий суперпользователя `root`. Такой подход широко используется разработчиками и позволяет установить сразу несколько версий. При желании вы можете изменить каталог установки на `/usr/local/llvm`, собрав общесистемный пакет. Только не забудьте использовать команду `sudo`, когда будете создавать каталог для установки и выполнять команду `make install`. Ниже приводится используемая для этого последовательность команд:

```
$ mkdir каталог-для-установки
$ mkdir каталог-где-будет-выполняться-сборка
$ cd каталог-где-будет-выполняться-сборка
```

В этом разделе мы создадим отдельный каталог, где будут храниться объектные файлы, то есть, промежуточные результаты компиляции. Не выполняйте сборку в том же каталоге, где хранятся исходные тексты. Используйте следующие команды с параметрами, описанными в предыдущем разделе:

```
$ /PATH_TO_SOURCE/configure --disable-optimized
--prefix=../каталог-для-установки
$ make && make install
```

Совет. Желаящие могут использовать команду `make -jN`, чтобы позволить ей задействовать `N` экземпляров компилятора, которые будут работать параллельно, что повысит общую скорость процедуры сборки. Например, попробуйте команду `make -j4` (или укажите чуть большее число), если процессор вашего компьютера имеет четыре ядра.

Подождите, пока компиляция и установка всех компонентов не завершится. Обратите внимание, что сценарии сборки также обрабатывают другие репозитории, загруженные и сохраненные в дерево каталогов с исходными текстами LLVM – нам не требуется отдельно настраивать Clang или дополнительные инструменты Clang.

Чтобы убедиться в успехе сборки, всегда полезно после ее окончания выполнить команду `echo $?`. Пара символов `?` в командной оболочке – это имя переменной, хранящей код завершения последней команды, выполненной в текущем сеансе, а команда `echo` выводит значение этой переменной на экран. Поэтому очень важно выполнять данную команду сразу вслед за командой `make`. Если сборка прошла успешно, команда `make` вернет в переменной `?` значение 0, как и любая другая программа, успешно завершившая работу:

```
$ echo $?  
0
```

Настройте свою переменную окружения `PATH`, чтобы упростить доступ к только что установленным выполняемым файлам, и выполните первый тест, спросив у Clang номер версии:

```
$ export PATH="$PATH:каталог-для-установки/bin"  
$ clang -v  
clang version 3.4
```

С использованием CMake и Ninja

LLVM предлагает альтернативную, кросс-платформенную систему сборки на основе CMake, вместо традиционных сценариев `configure`. CMake может генерировать специализированные файлы `Makefile` для вашей платформы, чем-то напоминая сценарий `configure`, но CMake – более гибкая система и, способная также генерировать файлы сборки для других систем, таких как Ninja, Xcode и Visual Studio.

С другой стороны, Ninja – это небольшая и быстрая система сборки, замещающая систему GNU Make и связанные с ней файлы `makefile`. Если вам интересно узнать причины появления системы Ninja и историю ее развития, посетите страницу <http://aosabook>.

org/en/posa/ninja.html. Систему CMake можно настроить на создание файлов сборки для Ninja вместо файлов makefile, что позволяет использовать либо связку CMake и GNU Make, либо CMake и Ninja.

Использование последней связки дает преимущество более быстрой пересборки после внесения изменений в исходные тексты LLVM. Это особенно удобно для тех, кто занимается разработкой инструментов и расширений в дереве исходных текстов LLVM и использует систему сборки LLVM для компиляции своих проектов.

Проверьте, установлены ли у вас CMake и Ninja. Например, в системе Ubuntu выполните следующую команду:

```
$ sudo apt-get install cmake ninja-build
```

Кроме того, сборка LLVM с применением CMake дает возможность определять множество параметров настройки. Полный их список можно найти по адресу: <http://llvm.org/docs/CMake.html>. Ниже приводится перечень параметров, соответствующих параметрам, перечисленным выше. Эти параметры имеют те же значения по умолчанию, что и предыдущие параметры сценария `configure`:

- `СМАКЕ_BUILD_TYPE`: Строка, значение которой определяет тип сборки Release (выпуск) или Debug (отладочная). Тип сборки Release эквивалентен параметру `--enable-optimized` сценария `configure`, а тип Debug – параметру `--disable-optimized`.
- `СМАКЕ_ENABLE_ASSERTIONS`: Булево значение, соответствующее параметру `--enable-assertions` сценария `configure`.
- `BUILD_SHARED_LIBS`: Булево значение, соответствующее параметру `--enable-shared` сценария `configure`. Определяет, как должны компилироваться библиотеки – как разделяемые (`true`) или как статические (`false`). Разделяемые библиотеки не поддерживаются на платформе Windows.
- `СМАКЕ_INSTALL_PREFIX`: Строка, соответствующая параметру `--prefix-configure`. Определяет путь к каталогу установки.
- `LLVM_TARGETS_TO_BUILD`: Список целей для сборки, разделенных точками с запятой, примерно соответствует списку целей, разделенных запятыми, в параметре `--enable-targets` сценария `configure`.

Для определения значений параметров в команде `cmake` используется синтаксис `-DPARAMETER=value`.

Сборка в Unix с использованием CMake и Ninja

Воспроизведем тот же пример, что был показан выше, с применением сценария `configure`, но на этот раз выполним сборку с использованием CMake и Ninja:

Сначала создадим каталог и для сборки и установки:

```
$ mkdir каталог-где-будет-выполняться-сборка
$ mkdir каталог-для-установки
$ cd каталог-где-будет-выполняться-сборка
```

Напоминаю, что для сборки следует использовать каталог, отличный от каталога с исходными файлами LLVM. Далее запустим CMake с набором выбранных параметров:

```
$ cmake /каталог-с-исходными-текстами -G Ninja -DCMAKE_BUILD_TYPE="Debug"
-DCMAKE_INSTALL_PREFIX=" ../каталог-для-установки"
```

Вместо `/каталог-с-исходными-текстами` следует подставить абсолютный путь к каталогу с исходными текстами LLVM. Если вы пожелаете использовать традиционный путь на основе GNU Makefiles, параметр `-G Ninja` можно опустить. Теперь можно завершить сборку командой `ninja` или `make`, в зависимости от выбранного инструмента сборки. Если вы решили использовать `ninja`, выполните следующую команду:

```
$ ninja && ninja install
```

Если вы решили использовать `make`:

```
$ make && make install
```

Так же как в более раннем примере, проверить успешность компиляции можно с помощью простой команды. Помните, что она должна запускаться непосредственно после команды сборки, то есть, никакие другие команды не должны выполняться между ними, потому что они тоже сохраняют код завершения в переменной `$?` :

```
$ echo $?
0
```

Если предыдущая команда выведет ноль, значит сборка была выполнена успешно. В заключение настройте свою переменную окружения `PATH` и воспользуйтесь новым компилятором:

```
$ export PATH=$PATH:where-you-want-to-install/bin
$ clang -v
```

Устранение ошибок сборки

Если команда сборки вернула ненулевое значение, это означает что возникла ошибка. В этом случае Make или Ninja выведет на экран со-

общение. Найдите самую первую ошибку в выводе команды. Ошибки сборки стабильной версии LLVM обычно возникают, когда система не соответствует требованиям к версиям программных компонентов. Наиболее часто ошибки возникают из-за использования устаревшего компилятора. Например, попытка собрать LLVM 3.4 с помощью GNU g++ Version 4.4.3 закончится следующей ошибкой, которая появится уже после того, как большая половина исходных файлов LLVM будет скомпилирована:

```
[1385/2218] Building CXX object projects/compiler-rt/lib/interception/
CMakeFiles/RTInterception.i386.dir/interception_type_test.cc.o

FAILED: /usr/bin/c++ (...)_test.cc.o -c /local/llvm-3.3/llvm/projects/
compiler-rt/lib/interception/interception_type_test.cc

test.cc:28: error: reference to 'OFF64_T' is ambiguous

interception.h:31: error: candidates are: typedef __sanitizer::OFF64_T
OFF64_T

sanitizer_internal_defs.h:80: error: typedef __sanitizer::u64
__sanitizer::OFF64_T
```

Чтобы исправить эту проблему, можно было бы внести необходимые исправления в исходный код LLVM (как это сделать, можно узнать после недолгих поисков в Интернете или заглянув в исходные тексты), но едва ли вам хотелось бы исправлять каждую версию LLVM, которую понадобится скомпилировать. Намного проще и правильнее обновить компилятор.

Вообще, когда при сборке стабильной версии возникают ошибки, в первую очередь следует подумать о том, чем параметры вашей системы отличаются от рекомендуемых. Помните, что стабильные версии тщательно тестируются на нескольких платформах. С другой стороны, если вы пытаетесь выполнить сборку исходных текстов нестабильной версии из репозитория SVN, вполне возможно, что ошибки были внесены в результате одного из последних изменений, и тогда можно попробовать откатиться к более ранней версии из SVN.

Использование других подходов в Unix

В некоторых системах Unix имеются диспетчеры пакетов, которые автоматически собирают и устанавливают приложения из исходных текстов. Они поддерживают собственные способы и приемы компиляции, хорошо отлаженные и обеспечивающие автоматическое раз-

решение зависимостей. Далее мы попробуем оценить такие платформы в контексте сборки и установки LLVM и Clang:

- В Mac OS X имеется диспетчер пакетов *MacPorts*, который можно задействовать следующей командой:

```
$ port install llvm-3.4 clang-3.4
```

- В Mac OS X имеется также диспетчер пакетов *Homebrew*, который можно вызвать так:

```
$ brew install llvm -with-clang
```

- В FreeBSD 9.1 поддерживается система *портов* (ports), которой можно воспользоваться, как показано ниже (имейте в виду, что начиная с версии FreeBSD 10, Clang является компилятором по умолчанию, то есть, он уже установлен):

```
$ cd /usr/ports/devel/llvm34
```

```
$ make install
```

```
$ cd /usr/ports/lang/clang34
```

```
$ make install
```

- Ниже показано, как произвести сборку и установку в Gentoo Linux:

```
$ emerge sys-devel/llvm-3.4 sys-devel/clang-3.4
```

Windows и Microsoft Visual Studio

Порядок компиляции LLVM и Clang в Microsoft Windows мы покажем на примере использования Microsoft Visual Studio 2012 в Windows 8. Выполните следующие шаги:

1. Установите Microsoft Visual Studio 2012.
2. Установите официальный дистрибутив инструментов CMake, который можно получить на сайте <http://www.cmake.org>. В процессе установки не забудьте отметить флажок **Add CMake to the system PATH for all users** (Добавить CMake в системную переменную PATH).
3. CMake может сгенерировать файлы проекта, необходимые Visual Studio для настройки и сборки LLVM. Для этого сначала запустите инструмент `cmake-gui`. Затем щелкните на кнопке **Browse Source...** (Выбрать исходный каталог) и выберите каталог с исходным кодом LLVM. Затем щелкните на кнопке **Browse Build** (Выбрать каталог сборки) и выберите каталог,

куда должны сохраняться файлы, сгенерированные компилятором CMake, как показано на рис. 1.1:

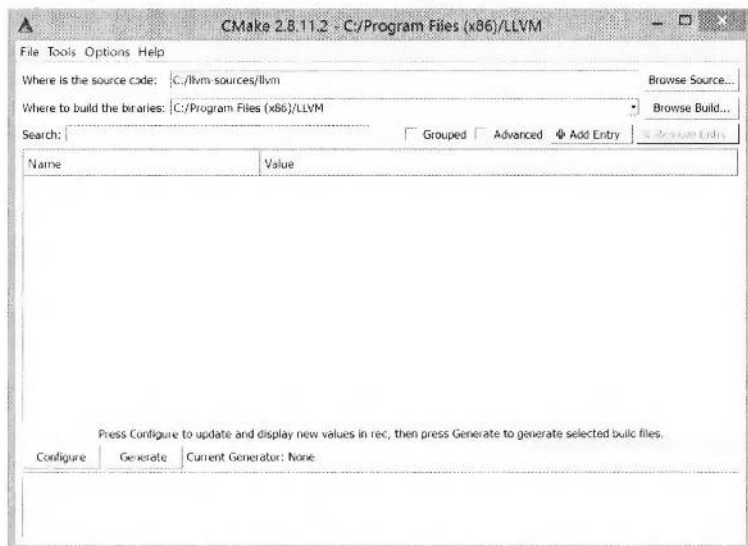


Рис. 1.1. Подготовка пакета LLVM к compilации

- Щелкните на кнопке **Add Entry** (Добавить элемент) и определите параметр `CMAKE_INSTALL_PREFIX`, содержащий путь установки инструментов LLVM, как показано на рис. 1.2:

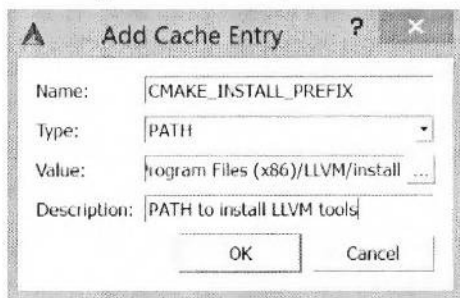


Рис. 1.2. Определение параметра `CMAKE_INSTALL_PREFIX`

- Дополнительно можно определить поддерживаемые цели в виде параметра `LLVM_TARGETS_TO_BUILD`, как показано на рис. 1.3. При необходимости можно добавить любые другие параметры для CMake, описанные выше.

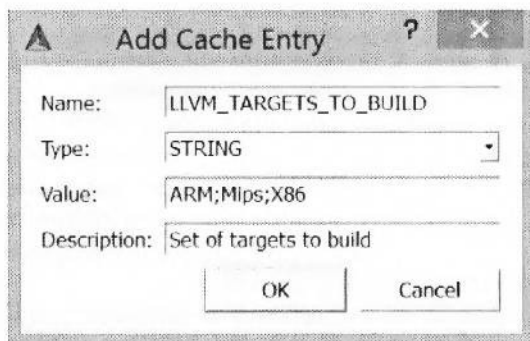


Рис. 1.3. Определение параметра LLVM_TARGETS_TO_BUILD

- Щелкните на кнопке **Configure** (Настроить). На экране появится диалог, предлагающий выбрать *генератор* для этого проекта и компилятор; выберите **Use default native compilers** (Использовать компиляторы по умолчанию) и для Visual Studio 2012, выберите в раскрывающемся списке пункт **Visual Studio 11**. Щелкните на кнопке **Finish** (Завершить), как показано на рис. 1.4:

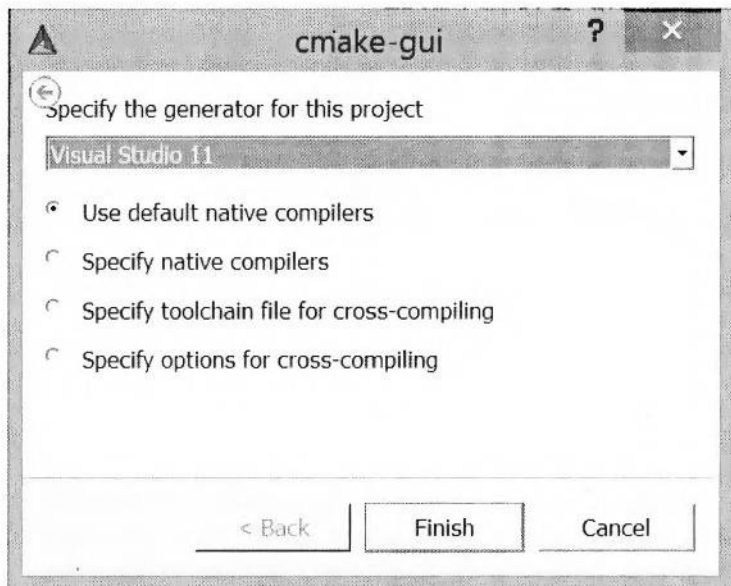


Рис. 1.4. Выбор генератора и компилятора

Совет. В Visual Studio 2013 используйте генератор для Visual Studio 12. Имя генератора соответствует версии среды разработки Visual Studio, а не ее коммерческому названию.

7. Определив настройки, щелкните на кнопке **Generate** (Сгенерировать). В результате будет создан файл решения Visual Studio, LLVM.sln и сохранен в указанный каталог сборки. Перейдите в этот каталог и дважды щелкните на файле – в Visual Studio откроется решение LLVM.
8. Чтобы автоматически собрать и установить LLVM/Clang, в панели обозревателя решения **Solution Explorer** найдите пункт CMakePredefinedTargets, распахни его, щелкните правой кнопкой мыши на пункте **INSTALL** и выберите пункт контекстного меню **Build** (Собрать). Предопределенная цель **INSTALL** указывает среде разработки, что она должна собрать и установить все инструменты и библиотеки LLVM/Clang (см. рис. 1.5):



Рис. 1.5. Сборка и установка LLVM/Clang

9. Чтобы выборочно собрать и установить определенные инструменты и/или библиотеки, выберите соответствующий элемент в панели обозревателя решения, щелкните на нем правой кнопкой мыши и выберите пункт контекстного меню **Build** (Собрать).
10. Добавьте каталог установки LLVM в системную переменную окружения PATH.

В нашем примере выбран каталог установки C:\Program Files (x86)\LLVM\install\bin. Чтобы проверить установку без изменения переменной окружения PATH, выполните следующую команду в окне командной строки:

```
C:>"C:\Program Files (x86)\LLVM\install\bin\clang.exe" -v  
clang version 3.4...
```

Mac OS X и Xcode

Скомпилировать пакет LLVM в Mac OS X можно, следуя инструкциям для Unix, что приводилось выше, однако то же самое можно сделать с помощью Xcode:

1. Установите Xcode.
2. Установите официальный дистрибутив инструментов CMake, который можно получить на сайте <http://www.cmake.org>. В процессе установки не забудьте отметить флажок **Add CMake to the system PATH for all users** (Добавить CMake в системную переменную PATH).
3. CMake может сгенерировать файлы проекта для Xcode. Для этого сначала запустите инструмент `cmake-gui`. Затем, как показано на рис. 1.6, щелкните на кнопке **Browse Source...** (Выбрать исходный каталог) и выберите каталог с исходным кодом LLVM. Затем щелкните на кнопке **Browse Build** (Выбрать каталог сборки) и выберите каталог, куда должны сохраняться файлы, сгенерированные компилятором CMake.
4. Щелкните на кнопке **Add Entry** (Добавить элемент) и определите параметр `CMAKE_INSTALL_PREFIX`, содержащий путь установки инструментов LLVM, как показано на рис. 1.7.
5. Дополнительно можно определить поддерживаемые цели в виде параметра `LLVM_TARGETS_TO_BUILD`, как показано на рис. 1.8. При необходимости можно добавить любые другие параметры для CMake, описанные выше.

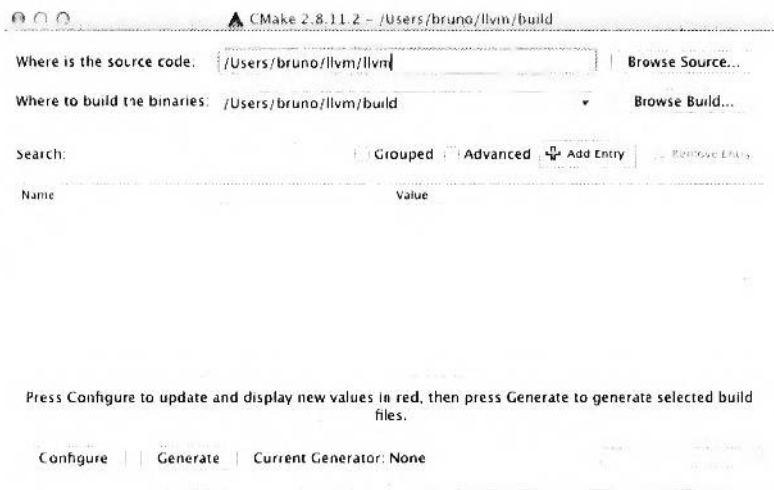


Рис. 1.6. Подготовка пакета LLVM к compilации

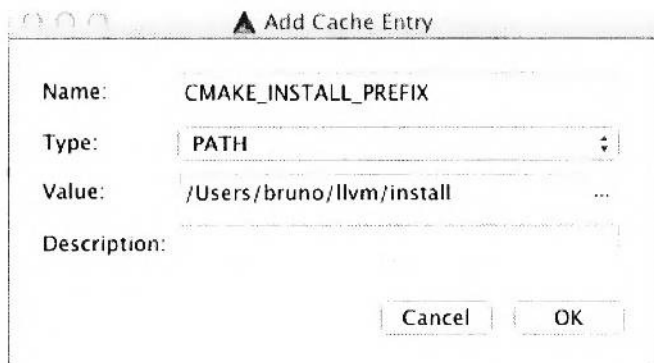


Рис. 1.7. Определение параметра CMAKE_INSTALL_PREFIX

6. Xcode не поддерживает создание **позиционно-независимого программного кода (Position Independent Code, PIC)** для библиотек LLVM. Поэтому, щелкните на кнопке **Add Entry** (Добавить элемент) и определите параметр `LLVM_ENABLE_PIC` типа **BOOL**, оставив сброшенным флажок **Value** (Значение), как показано на рис. 1.9.

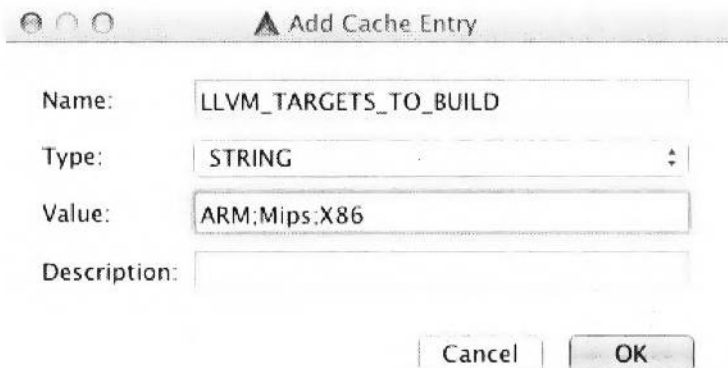


Рис. 1.8. Определение параметра LLVM_TARGETS_TO_BUILD

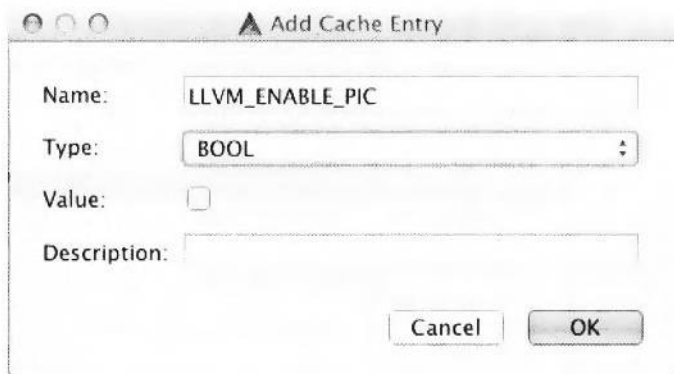


Рис. 1.9. Определение параметра LLVM_ENABLE_PIC

- Щелкните на кнопке **Configure** (Настроить). На экране появится диалог, предлагающий выбрать генератор для этого проекта и компилятор. Выберите **Use default native compilers** (Использовать компиляторы по умолчанию) и Xcode. Щелкните на кнопке **Finish** (Завершить), как показано на рис. 1.10.
- Определив настройки, щелкните на кнопке **Generate** (Сгенерировать). В результате этого будет создан файл `LLVM.xcodeproj` и сохранен в указанный каталог сборки. Перейдите в этот каталог и дважды щелкните на файле – в Xcode откроется проект LLVM.

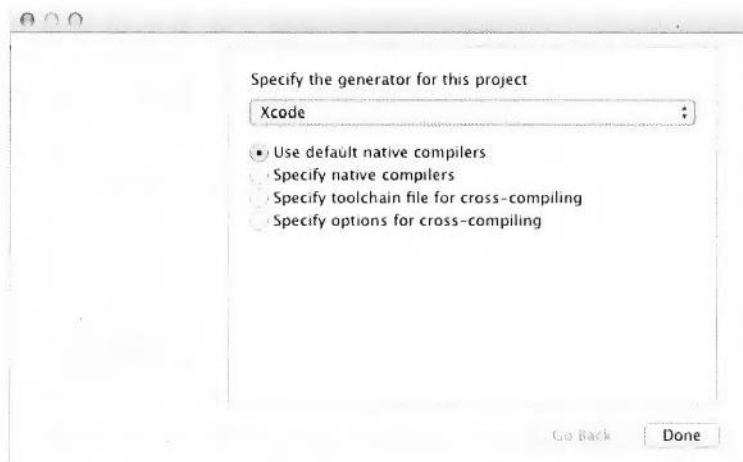


Рис. 1.10. Выбор генератора и компилятора

9. Чтобы автоматически собрать и установить LLVM/Clang, выберите схему **install** (установить).

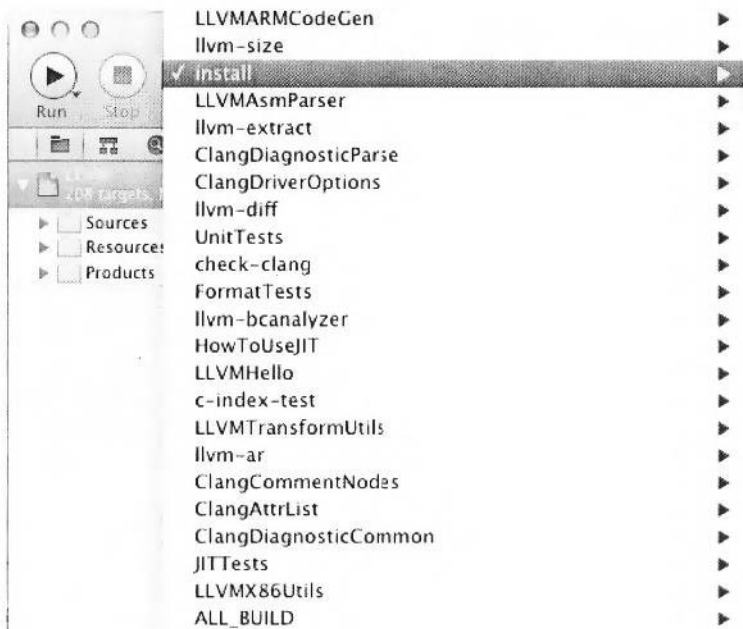


Рис. 1.11. Выбор схемы install

10. Затем откройте меню Product (Продукт) и выберите пункт **Build** (Собрать), как показано на рис. 1.12:

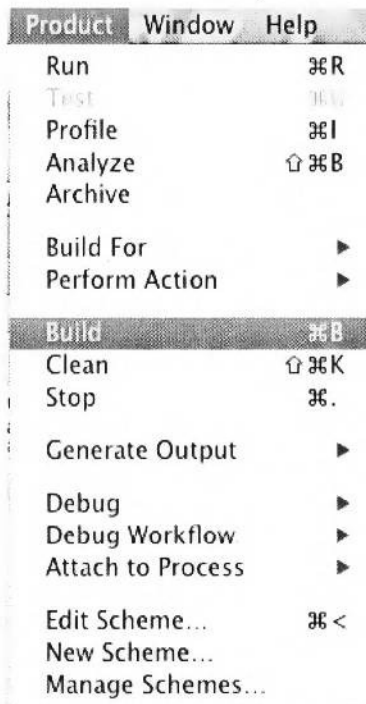


Рис. 1.12. Сборка и установка LLVM/Clang

11. Добавьте каталог установки LLVM в системную переменную окружения PATH.

В нашем примере выбран каталог установки `/Users/Bruno/llvm/install/bin`. Чтобы проверить установку без изменения переменной окружения PATH, выполните следующую команду, находясь в этом каталоге:

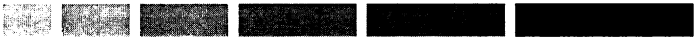
```
$ /Users/Bruno/llvm/install/bin/clang -v  
clang version 3.4...
```

В заключение

В этой главе были представлены подробные инструкции по установке LLVM и Clang из официальных скомпилированных пакетов, с при-

менением сторонних диспетчеров пакетов. Кроме того, мы подробно рассмотрели порядок сборки пакетов из исходных текстов, с использованием стандартных инструментов Unix и сред разработки в разных операционных системах.

В следующей главе рассказывается, как установить другие проекты, основанные на LLVM, которые могут пригодиться вам в ваших исследованиях. Эти внешние проекты обычно реализуют инструменты, созданные за рамками проекта LLVM, и распространяются отдельно.



ГЛАВА 2. Внешние проекты

Проекты, развиваемые отдельно от основных проектов LLVM и Clang, должны загружаться и устанавливаться отдельно. В этой главе мы познакомим вас с некоторыми другими проектами LLVM и расскажем, как собрать и установить их. Читатели, интересующиеся только основными инструментами LLVM, могут пропустить эту главу и вернуться к ней, когда появится такая необходимость.

В этой главе мы расскажем о следующих проектах:

- ◆ дополнительные инструменты Clang;
- ◆ Compiler-RT;
- ◆ DragonEgg;
- ◆ пакет тестов LLVM;
- ◆ LLDB;
- ◆ libc++.

Помимо проектов, описываемых в этой главе, существует также два официальных проекта LLVM, не вошедших в данную книгу: Polly, полиэдральный оптимизатор, и lld, компоновщик LLVM, который в настоящее время находится в разработке.

Предварительно скомпилированные пакеты не включают никакие внешние проекты, представленные в этой главе, кроме Compiler-RT. Поэтому здесь, в отличие от предыдущей главы, мы будем рассматривать только приемы сборки и установки из исходных текстов.

Не следует ждать от всех этих проектов такой же зрелости и надежности, как от основного проекта LLVM/Clang. Некоторые из них имеют статус экспериментальных или находятся на начальном этапе развития.

Введение в дополнительные инструменты Clang

Наиболее заметным архитектурным решением в проекте LLVM является выделение генератора выполняемого кода и анализатора исход-

ного кода (backend и frontend) в два отдельных проекта, ядро LLVM и Clang. Проект LLVM начинался как комплект инструментов вокруг промежуточного представления LLVM (Intermediate Representation, IR), опиравшихся на GCC для трансляции программ на языке высокого уровня в их промежуточное представление IR, в виде файлов *биткода* (bitcode). Термин «биткод» был придуман в пику термину «байткод», используемому разработчиками Java. Важной вехой в развитии проекта LLVM стало появление Clang – первого анализатора исходного кода, спроектированного командой LLVM, имеющего тот же уровень качества, ясную документацию и организацию библиотек, что и ядро LLVM. Он способен не только транслировать программы на C и C++ в LLVM IR (промежуточное представление LLVM), но также контролировать весь процесс преобразования, как гибкий компилятор, в значительной степени совместимый с GCC.

С этого момента мы будем называть Clang не драйвером компилятора, а анализатором исходного кода, ответственным за трансляцию программ на C и C++ в LLVM IR. Интереснейшим аспектом библиотек Clang является возможность их использования для создания мощных инструментов, дающих программистам на C++ свободу выполнения различных манипуляций с программным кодом на C++, таких как рефакторинг и анализ исходного кода. В состав проекта Clang входит несколько дополнительных инструментов, глядя на которые нетрудно понять, какие возможности дарят эти библиотеки:

- **Clang Check:** дает возможность выполнять проверку синтаксиса, применять исправления типичных проблем и выводить любые программы в виде абстрактного синтаксического дерева (Abstract Syntax Tree, AST);
- **Clang Format:** включает инструмент и библиотеку LibFormat, способные оформлять отступы и форматировать любые фрагменты исходного кода в соответствии со стандартами LLVM, а также рекомендациями по оформлению Google (Google Style Guide), Chromium (Chromium Style Guide), Mozilla (Mozilla Style Guide) и WebKit (WebKit Style Guide).

Репозиторий `clang-tools-extra` – это коллекция множества приложений, построенных на основе Clang. Они способны обрабатывать значительные объемы исходных текстов на C или C++ и выполнять все виды рефакторинга и анализа. Ниже перечислены некоторые из инструментов, доступных в пакете:

- **Clang Modernizer:** инструмент для рефакторинга, который сканирует исходный код на C++ и изменяет старые конструк-

ции так, чтобы они соответствовали более современным стилям программирования, которые определяются новейшими стандартами, такими как C++-11;

- **Clang Tidy**: инструмент, проверяющий наличие в исходном коде распространенных нарушений стандартов оформления, принятых в LLVM или Google;
- **Modularize**: помогает выявить заголовочные файлы C++, пригодные для составления модуля, новой концепции, в настоящее время обсуждаемой в комитете по стандартизации C++ (за дополнительной информацией обращайтесь к главе 10, «Инструменты Clang и фреймворк LibTooling»);
- **PPTrace**: простой инструмент слежения за деятельностью препроцессора Clang C++.

Дополнительную информацию о пользовании этими инструментами и создании собственных инструментов вы найдете в главе 10, «Инструменты Clang и фреймворк LibTooling».

Сборка и установка дополнительных инструментов Clang

Получить исходные тексты официальной версии 3.4 этого проекта можно по адресу: <http://llvm.org/releases/3.4/clang-tools-extra-3.4.src.tar.gz>. Если вам интересно увидеть все доступные версии, зайдите на страницу <http://llvm.org/releases/download.html>. Чтобы скопировать этот комплект инструментов без лишних сложностей, собирайте их вместе с исходными текстами LLVM и Clang, с помощью системы сборки LLVM. Для этого каталог с исходными текстами инструментов следует скопировать в дерево исходных текстов Clang, как показано ниже:

```
$ wget http://llvm.org/releases/3.4/clang-tools-extra-3.4.src.tar.gz
$ tar xzf clang-tools-extra-3.4.src.tar.gz
$ mv clang-tools-extra-3.4 llvm/tools/clang/tools/extra
```

Исходный код инструментов можно также получить непосредственно из официального репозитория LLVM:

```
$ cd llvm/tools/clang/tools
$ svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

Как уже говорилось в предыдущей главе, `trunk` можно заменить на `tags/RELEASE_34/final`, если требуется получить исходный код стабильной версии 3.4. Если вы предпочитаете использовать систему

управления версиями GIT, вы сможете получить исходный код, выполнив следующие команды:

```
$ cd llvm/tools/clang/tools
$ git clone http://llvm.org/git/clang-tools-extra.git extra
```

После копирования файлов в дерево исходных текстов Clang необходимо выполнить действия, предписываемые инструкциями в главе 1, «Сборка и установка LLVM», либо с помощью CMake, либо с помощью сценария configure. Проверить успешность сборки можно с помощью инструмента clang-modernize:

```
$ clang-modernize -version
clang-modernizer version 3.4
```

Compiler-RT

Проект Compiler-RT обеспечивает низкоуровневую функциональность для целей, которая не поддерживается аппаратной архитектурой. Например, при компиляции 32-разрядных целей, машинные инструкции деления 64-разрядных чисел обычно недоступны. Compiler-RT решает эту проблему, предоставляя специализированную, в зависимости от цели, и оптимизированную функцию, которая реализует деление 64-разрядных чисел с использованием 32-разрядных инструкций. Он обеспечивает поддержку тех же функциональных возможностей, что и libgcc. Кроме того, проект включает поддержку инструментов динамического тестирования адресов и памяти. Загрузить Compiler-RT Version 3.4 можно по адресу: <http://llvm.org/releases/3.4/compiler-rt-3.4.src.tar.gz>. Полный перечень доступных версий вы найдете на странице <http://llvm.org/releases/download.html>.

Так как этот компонент выполняет важные функции в компиляторе на основе LLVM, в предыдущей главе мы уже показали, как установить Compiler-RT. Если он у вас пока не установлен, не забудьте добавить его исходные тексты в каталог projects, внутри дерева каталогов с исходными текстами LLVM, например, выполнив следующую последовательность команд:

```
$ wget http://llvm.org/releases/3.4/compiler-rt-3.4.src.tar.gz
$ tar xzf compiler-rt-3.4.src.tar.gz
$ mv compiler-rt-3.4 llvm/projects/compiler-rt
```

Желающие могут извлечь исходные тексты из репозитория SVN:

```
$ cd llvm/projects
$ svn checkout http://llvm.org/svn/llvm-project/compiler-rt/trunk
compiler-rt
```

Исходные тексты можно также получить из GIT-зеркала:

```
$ cd llvm/projects
$ git clone http://llvm.org/git/compiler-rt.git
```

Примечание. *Compiler-RT* работает на множестве платформ, таких как GNU/Linux, Darwin, FreeBSD и NetBSD. Поддерживает аппаратные архитектуры: i386, x86_64, PowerPC, SPARC64 и ARM.

Compiler-RT в действии

Чтобы увидеть, как действует Compiler-RT, проведем простой эксперимент, написав программу на языке C, которая выполняет 64-разрядное деление:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
int main() {
    uint64_t a = 0ULL, b = 0ULL;
    scanf ("%lld %lld", &a, &b);
    printf ("64-bit division is %lld\n", a / b);
    return EXIT_SUCCESS;
}
```

Совет. Загрузка исходного кода примеров. Исходные тексты всех примеров можно загрузить на сайте издательства www.dmkpress.com или www.dmk.ru, в разделе «Читателям – Файлы к книгам».

Если в вашем распоряжении имеется система с аппаратной архитектурой x86_64, выполните компиляцию двумя способами:

```
$ clang -S -m32 test.c -o test-32bit.S
$ clang -S test.c -o test-64bit.S
```

Флаг `-m32` сообщает компилятору, что он должен сгенерировать программу для 32-разрядной архитектуры x86, а флаг `-s` требует произвести файл `test-32bit.S` с текстом программы на языке ассемблера x86.

Если заглянуть внутрь этого файла, можно увидеть необычный вызов, выполняющий деление:

```
call ___udivdi3
```

Данная функция определена в `Compiler-RT` и наглядно показывает, как используется эта библиотека. Но, если опустить флаг `-m32` и использовать 64-разрядный компилятор для архитектуры `x86`, как показано во второй команде, будет сгенерирован файл с исходным текстом на языке ассемблера `test-64bit.S` и в нем вы не увидите вызовы вспомогательных функций из `Compiler-RT`, потому что 64-разрядное деление в архитектуре `x86_64` выполняется единственной машинной инструкцией:

```
divq -24(%rbp)
```

Расширение DragonEgg

Как уже говорилось выше, LLVM начинался как проект, основанный на GCC, – в нем отсутствовал собственный анализатор исходного кода C/C++. На ранних этапах развития проекта LLVM требовалось загрузить доработанные исходные тексты GCC, распространявшиеся в виде пакета `llvm-gcc`, и компилировать всю связку вместе. Так как при этом требовалось скомпилировать весь пакет GCC, эта задача отнимала много времени и сама по себе была очень сложной, требующей знаний об особенностях сборки. Позднее появился проект DragonEgg, использовавший в своих интересах систему расширений (плагинов) GCC и отделивший логику LLVM в собственное дерево исходных текстов, имеющее намного меньшие размеры. Благодаря ему пользователям требовалось пересобрать уже не весь пакет GCC, а только расширение, и затем подключать его к GCC. Кроме того, проект DragonEgg является единственным в LLVM, исходные тексты которого распространяются на условиях лицензии GPL.

Даже после появления проекта Clang, DragonEgg сохранил свои позиции, потому что Clang поддерживает только языки C и C++, тогда как GCC способен обрабатывать программы на самых разных языках. Благодаря расширению DragonEgg, вы можете использовать GCC как анализатор исходного кода к компилятору LLVM, и компилировать программы, написанные на большинстве языков, поддерживаемых GCC: Ada, C, C++ и FORTRAN, с частичной поддержкой Go, Java, Obj-C и Obj-C++.

Действие расширения заключается в замене компонентов компилятора GCC соответствующими компонентами LLVM и автоматическом выполнении всех этапов компиляции, как и полагается первоклассному драйверу компилятора. Конвейер компиляции в этой ситуации представлен на рис. 2.1.

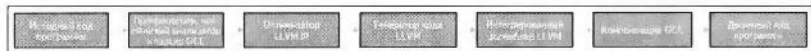


Рис. 2.1. Конвейер компиляции после внедрения DragonEgg

При желании можно использовать флаги `-fplugin-arg-dragonegg-emit-ir -S`, чтобы прервать компиляцию после этапа создания LLVM IR и с помощью инструментов LLVM исследовать результаты работы анализатора исходного кода или завершить компиляцию вручную. Пример такого подхода будет представлен ниже.

Так как DragonEgg является побочным проектом, его дистрибутивы обновляются не одновременно с главным проектом LLVM. Например, на момент написания этих строк последней была версия DragonEgg 3.3, связанная с версией LLVM 3.3. Соответственно, биткод LLVM, полученный с помощью LLVM IR, нельзя будет проанализировать, оптимизировать или скомпилировать с применением инструментов LLVM, если их версия отличается от версии 3.3. Официальный веб-сайт проекта DragonEgg находится по адресу: <http://dragonegg.llvm.org>.

Сборка DragonEgg

Чтобы скомпилировать и установить DragonEgg, сначала необходимо получить исходные тексты, по адресу: <http://llvm.org/releases/3.3/dragonegg-3.3.src.tar.gz>. Для этого в Ubuntu можно выполнить следующие команды:

```
$ wget http://llvm.org/releases/3.3/dragonegg-3.3.src.tar.gz
$ tar xzvf dragonegg-3.3.src.tar.gz
$ cd dragonegg-3.3.src
```

Желающие могут получить исходные тексты из SVN:

```
$ svn checkout http://llvm.org/svn/llvm-project/dragonegg/trunk
dragonegg
```

Чтобы получить исходные тексты из GIT-зеркала, выполните следующие команды:

```
$ git clone http://llvm.org/git/dragonegg.git
```

Чтобы выполнить компиляцию и установку, необходимо указать каталог для установки. Версия LLVM должна совпадать с версией DragonEgg. Допустим, что нам требуется выполнить установку в тот же каталог, `/usr/local/llvm`, который использовался в главе 1, «Сборка и установка LLVM». Также предположим, что в системе

установлен компилятор GCC 4.6 и путь к нему присутствует в переменной окружения PATH. Сборку и установку теперь можно выполнить следующими командами:

```
$ GCC=gcc-4.6 LLVM_CONFIG=/usr/local/llvm/bin/llvm-config make  
$ cp -a dragonegg.so /usr/local/llvm/lib
```

Обратите внимание, что в проекте отсутствуют сценарий configure и файлы проекта для CMake. По этой причине сборка должна выполняться непосредственным вызовом команды make. Если команда gcc в вашей системе уже связана с требуемой версией GCC, префикс GCC=gcc-4.6 можно опустить. Расширение представляет собой разделяемую библиотеку с именем dragonegg.so, которую можно указывать в командной строке вызова GCC, как показано ниже. Давайте попробуем скомпилировать классическую программу «Hello, World!» на языке C.

```
$ gcc-4.6 -fplugin=/usr/local/llvm/lib/dragonegg.so hello.c -o hello
```

Совет. Теоретически расширение DragonEgg поддерживает GCC, начиная с версии 4.5, однако мы настоятельно рекомендуем использовать GCC 4.6, потому что с другими версиями GCC расширение DragonEgg протестировано не так тщательно.

Конвейер компиляции с применением DragonEgg и инструментов LLVM

Чтобы увидеть результат работы анализатора исходного кода, воспользуйтесь флагами -S -fplugin-arg-dragoneggemit-ir, которые обеспечивают сохранение в файле промежуточного представления LLVM IR в удобочитаемом виде:

```
$ gcc-4.6 -fplugin=/usr/local/llvm/lib/dragonegg.so -S -fplugin-  
argdragonegg-emit-ir hello.c -o hello.ll  
  
$ cat hello.ll
```

Способность прерывать компиляцию после трансляции программы в промежуточное представление (IR) и сохранение промежуточного кода на диск является характерной особенностью LLVM. Большинство других компиляторов не поддерживают такой возможности. После знакомства с представлением LLVM IR программы можно

продолжить процесс компиляции с помощью разных инструментов LLVM. Следующая команда вызывает ассемблер, который преобразует промежуточный код программы из текстового представления в двоичное и сохраняет результат на диск:

```
$ llvm-as hello.ll -o hello.bc
$ file hello.bc
hello.bc: LLVM bitcode
```

При желании двоичное представление можно преобразовать обратно в текст, воспользовавшись специальным дизассемблером IR (`llvm-dis`). Следующий инструмент применяет платформонезависимые оптимизации и выводит информацию об успешных преобразованиях кода:

```
$ opt -stats hello.bc -o hello.bc
```

Флаг `-stats` является необязательным. Далее можно применить инструмент LLVM, выполняющий трансляцию на язык ассемблера для данной аппаратной платформы:

```
$ llc -stats hello.bc -o hello.S
```

Здесь так же флаг `-stats` является необязательным. Так как результатом работы этого инструмента является файл с исходным кодом на языке ассемблера, его можно скомпилировать с помощью ассемблера GNU `binutils` или ассемблера LLVM. Следующая команда вызывает ассемблер LLVM:

```
$ llvm-mc -filetype=obj hello.S -o hello.o
```

По умолчанию LLVM использует системный компоновщик, потому что проект компоновщика LLVM, `lld`, в настоящее время не интегрирован в основной проект LLVM. Соответственно, если в системе не установлен компоновщик `lld`, компиляцию можно завершить с помощью обычного драйвера компилятора, который автоматически активирует системный компоновщик:

```
$ gcc hello.o -o hello
```

Отметьте, что по соображениям производительности, драйвер компилятора LLVM никогда не сохраняет промежуточные представления программ на диск, за исключением объектных файлов. Он использует представление, хранящееся в памяти, и передаст его разным инструментам LLVM в процессе компиляции.

Пакет тестов LLVM

Пакет тестов LLVM содержит официальный комплект программ и тестов для тестирования компилятора LLVM. Он может пригодиться разработчикам LLVM для проверки оптимизаций и улучшений компилятора путем сборки и запуска таких программ. Если вы используете нестабильную версию LLVM или вносите свои изменения в исходные тексты LLVM и подозреваете, что что-то работает не так, как предполагалось, используя пакет тестов, вы легко сможете протестировать LLVM самостоятельно. Но имейте в виду, что в дереве с исходными текстами LLVM имеются также регрессионные и модульные тесты, которые можно выполнить простой командой `make check-all`. Описываемый здесь пакет тестов отличается от классических регрессионных и модульных тестов тем, что содержит всеобъемлющие тесты.

Пакет тестов LLVM следует разместить в дереве каталогов с исходными текстами LLVM, и система сборки LLVM сама обнаружит его. Исходные тексты для версии 3.4 доступны по адресу: <http://llvm.org/releases/3.4/test-suite-3.4.src.tar.gz>.

Загрузить их можно следующими командами:

```
$ wget http://llvm.org/releases/3.4/test-suite-3.4.src.tar.gz
$ tar xzf test-suite-3.4.src.tar.gz
$ mv test-suite-3.4 llvm/projects/test-suite
```

Если вы предпочитаете использовать последнюю, пусть и нестабильную, версию из SVN, выполните следующие команды:

```
$ cd llvm/projects
$ svn checkout http://llvm.org/svn/llvm-project/test-suite/trunk
testsuite
```

Чтобы получить исходные тексты из GIT-зеркала, выполните следующие команды:

```
$ cd llvm/projects
$ git clone http://llvm.org/git/llvm-project/test-suite.git
```

Чтобы воспользоваться пакетом тестов, требуется повторно создать файлы сборки LLVM. В данном конкретном случае нельзя использовать CMake – для работы с тестами нужен классический сценарий `configure`. Повторите этапы настройки, как было описано в главе 1, «Сборка и установка LLVM».

Пакет тестов включает множество файлов `Makefile`, которые реализуют тестирование. Имеется также возможность писать собственные

файлы Makefile, выполняющие нестандартные тесты. Поместите свой файл Makefile в каталог с исходными текстами пакета, следуя соглашению об именовании `llvm/projects/test-suite/TEST.<custom>.Makefile`, где вместо тега `<custom>` укажите название своего теста. Пример оформления файлов Makefile вы найдете в `llvm/projects/test-suite/TEST.example.Makefile`.

Совет. Прежде чем воспользоваться своим или стандартным измененным файлом Makefile, необходимо заново создать файлы сборки LLVM.

В процессе настройки, в дереве каталогов с объектными файлами LLVM создается каталог для пакета тестов, откуда будут запускаться программы и тесты. Чтобы выполнить, например, тест `TEST.example.Makefile`, перейдите в каталог с объектными файлами и выполните следующие команды:

```
$ cd your-llvm-build-folder/projects/test-suite
$ make TEST="example" report
```

Использование LLDB

Проект LLDB (Low Level Debugger – низкоуровневый отладчик) – это отладчик, встроенный в инфраструктуру LLVM, активно разрабатываемый и распространяемый в составе Xcode 5 для Mac OS X. Разработка отладчика началась в 2011 году, вне рамок проекта Xcode и к моменту написания этих строк пока не была выпущена стабильная его версия. Получить исходные тексты LLDB можно по адресу: <http://llvm.org/releases/3.4/lldb-3.4.src.tar.gz>. Подобно многим проектам, основанным на LLVM, его компиляцию легко встроить в систему сборки LLVM. Для этого просто скопируйте в каталог с исходными текстами инструментов LLVM, как показано ниже:

```
$ wget http://llvm.org/releases/3.4/lldb-3.4.src.tar.gz
$ tar xvf lldb-3.4.src.tar.gz
$ mv lldb-3.4 llvm/tools/lldb
```

Желающие могут получить исходные тексты из SVN:

```
$ cd llvm/tools
$ svn checkout http://llvm.org/svn/llvm-project/lldb/trunk lldb
```

Если вы предпочитаете использовать GIT-зеркало:

```
$ cd llvm/tools
$ git clone http://llvm.org/git/llvm-project/lldb.git
```

Примечание. Версия LLDB для GNU/Linux до сих пор находится в состоянии экспериментальной.

Прежде чем приступать к сборке LLDB, проверьте выполнение предварительных требований: наличие в системе Swig, libedit (только для Linux) и Python. В Ubuntu, например, удовлетворить эти зависимости можно с помощью команды:

```
$ sudo apt-get install swig libedit-dev python
```

Не забудьте также заново сгенерировать файлы сборки LLVM, чтобы обеспечить автоматическую компиляцию LLDB. Для этого выполните действия, предписываемые инструкциями в главе 1, «Сборка и установка LLVM».

Чтобы проверить успешность установки отладчика lldb, просто запустите его с флагом `-v`:

```
$ lldb -v
lldb version 3.4 ( revision )
```

Пример сеанса отладки с помощью LLDB

Чтобы показать, как пользоваться отладчиком LLDB, мы продемонстрируем сеанс отладки выполняемого файла Clang. Этот файл содержит множество символов C++, доступных для исследования. Если вы скомпилировали проект LLVM/Clang с параметрами по умолчанию, выполняемый файл Clang будет включать отладочную информацию. Это происходит при запуске сценария `configure` без флага `--enable-optimized` или когда CMake вызывается с параметром `-DCMAKE_BUILD_TYPE="Debug"`.

Знакомым с отладчиком GDB будет интересно увидеть таблицу (<http://lldb.llvm.org/lldb-gdb.html>) соответствий команд отладчиков GDB и LLDB.

Как и при использовании отладчика GDB, запуская LLDB необходимо указать в командной строке путь к выполняемому файлу, который требуется отладить:

```
$ lldb каталог-для-установки-llvm/bin/clang
```

```
Current executable set to 'каталог-для-установки-llvm/bin/clang'
(x86_64).
```

```
(lldb) break main
```

```
Breakpoint 1: where = clang'main + 48 at driver.cpp:293, address =
0x000000001000109e0
```

Чтобы начать отладку, передадим выполняемому файлу Clang аргумент командной строки, например `-v`, который должен вывести номер версии Clang:

```
(lldb) run -v
```

После того, как LLDB достигнет точки останова, можно организовать пошаговое выполнение строк C++ командой `next`. Как и GDB, отладчик LLDB принимает любые сокращения команд, например, `n` вместо `next`, при сохранении однозначности их интерпретации:

```
(lldb) n
```

Чтобы увидеть, как LLDB выводит объекты C++, дойдем до строки, следующей за объявлением объекта `argv` или `ArgAllocator`, и выведем его:

```
(lldb) n
(lldb) p ArgAllocator
(llvm::SpecificBumpPtrAllocator<char>) $0 = {
  Allocator = {
    SlabSize = 4096
    SizeThreshld = 4096
    DefaultSlabAllocator = (Allocator = llvm::MallocAllocator @
0x00007f85f1497f68)
    Allocator = 0x00000007ffffbfff200
    CurSlab = 0x00000000000000000
    CurPtr = 0x00000000000000000
    End = 0x00000000000000000
    BytesAllocated = 0
  }
}
```

Закончив отладку, завершите отладчик командой `q`:

```
(lldb) q
Quitting LLDB will kill one or more processes. Do you really
want to proceed: [Y/n] y
```

Введение в стандартную библиотеку `libc++`

Библиотека `libc++` – это стандартная библиотека C++, переписанная для проекта LLVM с целью поддержки последних стандартов C++, включая C++11 и C++1y, и распространяемая на условиях двух лицензий: MIT License и UIUC License. Библиотека `libc++` является важнейшим спутником `Compiler-RT` и входит в комплект библиотек времени выполнения, которые используются `Clang++` для сборки вы-

полняемых файлов программ на языке C++, наряду с библиотекой `libc` (библиотека времени выполнения `OpenCL`), когда это необходимо. Она не входит в состав проекта `Compiler-RT`, поэтому от вас не требуется компилировать `libc++`. `Clang` не ограничивается этой библиотекой и может компоновать программы с библиотекой `GNU libstdc++`, если `libc++` недоступна. Если у вас есть обе библиотеки, вы сможете выбирать, какую библиотеку использовать, указывая нужную в параметре `-stdlib`. Библиотека `libc++` поддерживает аппаратные архитектуры `x86` и `x86_64`, и была создана, как замена `GNU libstdc++` для систем `Mac OS X` и `GNU/Linux`.

Совет. *Версия библиотеки `libc++` для `GNU/Linux` все еще находится в стадии разработки и пока не настолько стабильна как версия для `Mac OS X`.*

Согласно мнению разработчиков, одно из основных препятствий, мешающих продолжать использовать `GNU libstdc++`, состоит в необходимости переписать большую часть кода для поддержки новейших стандартов C++ и в привязке `libstdc++` к лицензии `GPLv3`, которая не устраивает некоторые компании, использующие проект `LLVM` в своих разработках. Обратите внимание, что проекты `LLVM` широко используются в коммерческих продуктах, исповедующих философию, несовместимую с философией `GPL`. Перед лицом этих проблем сообщество `LLVM` решило создать новую стандартную библиотеку C++ для `Mac OS X` и с поддержкой `Linux`.

Самый простой способ получить библиотеку `libc++` в свое распоряжение на компьютере `Apple` – установить `Xcode` версии 4.2 или выше.

Если вы собираетесь компилировать библиотеку самостоятельно в системе `GNU/Linux`, учтите, что стандартная библиотека C++ состоит из собственно библиотеки и низкоуровневого слоя, реализующего обработку исключений и поддержку **определения типа во время выполнения (Run-Time Type Information, RTTI)**. Такое разделение упрощает перенос стандартной библиотеки C++ в другие системы и обеспечивает некоторые дополнительные возможности при сборке собственной стандартной библиотеки C++. Вы можете собрать `libc++`, скомпоновав ее с `libsupc++`, `GNU`-реализацией низкоуровневого слоя, или с `libc++abi`, реализацией, созданной командой `LLVM`. Однако, в настоящее время `libc++abi` поддерживает только системы `Mac OS X`.



Чтобы собрать `libc++` с `libsups++` в системе GNU/Linux, сначала необходимо загрузить пакеты с исходными текстами:

```
$ wget http://llvm.org/releases/3.4/libcxx-3.4.src.tar.gz
$ tar xvf libcxx-3.4.src.tar.gz
$ mv libcxx-3.4 libcxx
```

На момент написания этих строк все еще отсутствовала возможность использовать систему сборки LLVM для компиляции библиотеки. Поэтому, отметьте, что мы не копируем исходные тексты `libc++` в дерево каталогов LLVM.

Самую свежую версию исходных текстов можно получить из репозитория SVN:

```
$ svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx
```

Или из GIT-зеркала:

```
$ git clone http://llvm.org/git/llvm-project/libcxx.git
```

При наличии рабочего компилятора на основе LLVM, необходимо сгенерировать файлы сборки для `libc++`, использующие новый компилятор LLVM. В этом примере мы будем полагать, что у нас уже имеется действующий компилятор LLVM 3.4.

Чтобы использовать `libsups++`, сначала нужно узнать, где в системе находятся ее заголовочные файлы. Так как они являются частью обычного компилятора GCC для GNU/Linux, вы сможете найти их с помощью следующей команды:

```
$ echo | g++ -Wp,-v -x c++ - -fsyntax-only
```

```
порядок поиска для #include "...":
порядок поиска для #include <...>:
/usr/include/c++/4.7.0
/usr/include/c++/4.7.0/x86_64-pc-linux-gnu
(последующие элементы списка опущены)
```

Обычно первые два пути в списке соответствуют каталогам с заголовочными файлами библиотеки `libsups++`. Чтобы убедиться в этом, проверьте присутствие какого-нибудь заголовочного файла, характерного именно для `libsups++`, такого как `bits/exception_ptr.h`:

```
$ find /usr/include/c++/4.7.0 | grep bits/exception_ptr.h
```

Далее, сгенерируйте файлы сборки библиотеки `libc++` для ее компиляции с помощью компилятора на основе LLVM. Для этого переопределите переменные окружения `CC` и `CXX`, содержащие пути к системным компиляторам для языков C и C++, соответственно, и

укажите в них путь к компилятору LLVM, который должен использоваться для компиляции `libc++`. Чтобы собрать `libc++` и `libsupc++` с помощью CMake, нужно указать параметры `LIBCXX_CXX_ABI`, определяющий путь к библиотеке низкоуровневого слоя, и `LIBCXX_LIBSUPCXX_INCLUDE_PATHS`, содержащий список путей, разделенных точками с запятой, к каталогам с заголовочными файлами `libsupc++`, которые были только что обнаружены:

```
$ mkdir каталог-где-будет-выполняться-сборка
$ cd каталог-где-будет-выполняться-сборка
$ CC=clang CXX=clang++ cmake -DLIBCXX_CXX_ABI=libstdc++
-DLIBCXX_LIBSUPCXX_INCLUDE_PATHS="/usr/include/c++/4.7.0;/usr/
include/c++/4.7.0/x86_64-pc-linux-gnu" -DCMAKE_INSTALL_PREFIX=
"/usr" ../libcxx
```

На этом этапе убедитесь, что каталог `../libcxx` существует и хранит исходные тексты `libc++`. Выполните команду `make`, чтобы собрать проект. Для выполнения команд установки используйте `sudo`, потому что библиотека будет установлена в каталог `/usr`, чтобы `clang++` мог найти ее:

```
$ make && sudo make install
```

Вы можете поэкспериментировать с новой библиотекой и поддержкой новейших стандартов C++, используя флаг `-stdlib=libc++` при компиляции своих проектов компилятором `clang++`.

Чтобы увидеть новую библиотеку в действии, скомпилируйте простое приложение на C++ следующей командой:

```
$ clang++ -stdlib=libc++ hello.cpp -o hello
```

С помощью команды `readelf` легко можно убедиться, что выполняемый файл `hello` действительно скомпонован с новой библиотекой `libc++`:

```
$ readelf d hello
```

```
Dynamic section at offset 0x2f00 contains 25 entries:
```

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc++.so.1]

Данный список содержит и другие элементы, которые мы опустили. В первом же динамическом разделе ELF мы видим требование на загрузку только что скомпилированной разделяемой библиотеки

libc++.so.1. Это подтверждает, что вновь скомпилированные программы на C++ теперь будут использовать новую стандартную библиотеку C++ из LLVM. Дополнительную информацию ищите на официальном сайте проекта <http://libcxx.llvm.org/>.

В заключение

В рамках LLVM фактически развивается несколько проектов. Некоторые из них не требуются для работы основного драйвера компилятора, но содержат ценные инструменты и библиотеки. В этой главе мы показали, как собирать и устанавливать эти компоненты. В последующих главах мы подробнее исследуем некоторые из них. Мы советуем читателям периодически возвращаться к этой главе за инструкциями по установке.

В следующей главе мы познакомим вас с организацией основных библиотек и инструментов в проекте LLVM.

ГЛАВА 3.

Инструменты и организация

Проект LLVM включает несколько библиотек и инструментов, которые, все вместе, образуют большую инфраструктуру компилятора. Внимательное отношение к организации является ключом к слаженной работе всех компонентов. Красной нитью через проект LLVM проходит философия «все сущее есть библиотека», благодаря чему остается лишь малая часть кода, которая не может быть использована повторно и принадлежит тому или иному инструменту. Большое число инструментов дают пользователям возможность использовать библиотеки множеством способов. В этой главе рассматриваются следующие темы:

- ♦ обзор и организация основных библиотек LLVM;
- ♦ порядок работы драйвера компилятора;
- ♦ за рамками компилятора: знакомство с промежуточными инструментами LLVM;
- ♦ создание первого собственного инструмента LLVM;
- ♦ общие советы по навигации в исходных текстах LLVM.

Введение в основные принципы организации LLVM

LLVM – это удивительно поучительный фреймворк, организованный в виде множества инструментов и позволяющий любопытным пользователям изучать разные стадии компиляции. Организационная структура проекта была заложена в самых первых его версиях, более 10 лет тому назад, когда реализация, в значительной степени состоящая из алгоритмов трансляции в машинный код, опиралась на GCC для трансляции программ на языках высокого уровня, таких как C, в промежуточное представление LLVM (IR). Сегодня центральным аспектом LLVM является промежуточное представление IR в форме

Single-Static Assignments (SSA)¹, обладающее двумя важными характеристиками:

- код организован в виде трехадресных инструкций;
- имеет бесконечное множество регистров.

Однако это не означает, что LLVM использует единственную форму представления программ. В процессе компиляции создаются другие промежуточные структуры данных, хранящие логику программы и помогающие в ее трансляции. Технически, они так же являются промежуточными представлениями программ. Например, на разных стадиях компиляции LLVM использует следующие дополнительные структуры данных:

- когда программа на языке C или C++ транслируется в LLVM IR, Clang создает представление этой программы в памяти, в виде **абстрактного синтаксического дерева (Abstract Syntax Tree, AST)** – экземпляра класса `TranslationUnitDecl`;
- когда LLVM IR транслируется в исходный код на языке ассемблера, LLVM сначала преобразует программу в **ориентированный ациклический граф (Directed Acyclic Graph, DAG)**, чтобы упростить выборку инструкций (экземпляр класса `SelectionDAG`), и затем преобразует его обратно в трехадресное представление, чтобы обеспечить возможность трансляции в машинные команды (экземпляр класса `MachineFunction`);
- для реализации ассемблеров и компоновщиков LLVM использует четвертую промежуточную структуру данных (экземпляр класса `MCMODULE`) для хранения представлений программ в виде объектных файлов.

Среди всех форм представления программ в LLVM самой важной является LLVM IR. Программы в этом представлении могут существовать не только в памяти, но и сохраняться на диск. Возможность сохранения программ в представлении LLVM IR на диске является еще одним важным решением, принятым на ранних этапах развития проекта и отражает академический интерес к изучению концепции оптимизации программ на всем жизненном цикле (*lifelong program optimizations*).

В этой философии компилятор не применяет оптимизаций во время компиляции, оставляя возможность выполнения оптимизаций во время установки, выполнения и простоя (когда программа не работает). При таком подходе оптимизация выполняется в течение всего

¹ Когда каждой переменной значение присваивается лишь единожды. – Прим. перев.

жизненного цикла программы, что объясняет название концепции. Например, когда пользователь не работает с программой и компьютер простаивает, операционная система может запустить демон компилятора для профилирования данных, собранных во время выполнения, чтобы повторно оптимизировать программу под конкретные условия использования.

Обратите внимание, что поддержка сохранения LLVM IR на диск, которая обеспечивает возможность оптимизации программ на всем жизненном цикле, позволяет также разными способами кодировать целые программы. Когда программа целиком хранится в форме промежуточного представления IR, открывается возможность выполнения целого класса высокоэффективных межпроцедурных оптимизаций, пересекающих границы единиц трансляции или файлов C. Соответственно открывается возможность применения оптимизаций во время компоновки (связывания).

С другой стороны, чтобы оптимизация на всем жизненном цикле стала реальностью, программы должны распространяться в представлении LLVM IR. Это предполагает работу LLVM в роли платформы или виртуальной машины и конкуренцию с Java, а это серьезная проблема. Например, представление LLVM IR не является системно-независимым, как Java. Кроме того, разработчики LLVM еще не занимались оптимизациями с обратной связью (feedback-directed optimizations) для применения после установки. Читателям, интересующимся этой темой, мы предлагаем прочитать весьма информативное обсуждение на **LLVMdev**: <http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043719.html>.

По мере развития проекта, проектное решение о поддержке сохранения промежуточного представления на диск продолжало рассматриваться как возможность оптимизации на этапе компоновки, но все меньше внимания уделялось идее оптимизации на всем жизненном цикле. В конечном счете, участники проекта LLVM подтвердили отсутствие заинтересованности в его становлении платформой, заметив название Low Level Virtual Machine (низкоуровневая виртуальная машина) исторически сложившейся аббревиатурой LLVM, чем дали понять, что целью проекта LLVM является развитие мощного и практичного компилятора C/C++, а не конкуренция с Java.

Промежуточное представление на диске само по себе имеет перспективные применения, помимо оптимизации на этапе компоновки, которые некоторые группы стремятся привнести в реальный мир. Например, сообщество FreeBSD работает над созданием выполняемых



файлов программ в представлении LLVM, чтобы обеспечить возможность микроархитектурных оптимизаций во время установки или во время простоя. В этом сценарии, даже если программа была скомпилирована для обобщенной архитектуры x86, во время ее установки, например, на платформу Intel Haswell x86, инфраструктура LLVM могла бы использовать представление LLVM и специализировать выполняемый код программы для использования новых инструкций, поддерживаемых процессором Haswell. Даже при том, что эта новая идея пока находится в стадии оценки, она демонстрирует, что промежуточное представление LLVM на диске позволяет создавать радикально новые решения. На микроархитектурную оптимизацию возлагаются большие надежды, потому что полная независимость от платформы, как в Java, выглядит непрактичной в LLVM, и данная возможность в настоящее время пока исследуется только во внешних проектах (см. PNaCl, Chromium Portable Native Client).

Разработка основных библиотек компилятора IR велась в соответствии с двумя основными принципами LLVM IR:

- использование представления SSA и бесконечное число регистров, чтобы обеспечить быструю оптимизацию;
- упрощение оптимизации на этапе компоновки за счет сохранения на диске программ целиком в представлении IR.

LLVM сегодня

К настоящему времени в рамках проекта LLVM была создана и продолжает развиваться большая коллекция инструментов, имеющих отношение к компиляции. Фактически, под названием LLVM скрывается следующее:

- **Проект/инфраструктура LLVM:** Множество различных проектов, которые все вместе образуют законченный компилятор: анализаторы исходного кода (frontends), генераторы выполняемого кода (backends), оптимизаторы, ассемблеры, компоновщики, libc++, Compiler-RT и механизм динамической (Just-In-Time, JIT) компиляции. Такое значение слово «LLVM» имеет, например, в предложении: «LLVM – состоит из нескольких проектов».
- **Компилятор на основе LLVM:** Компилятор, частично или полностью опирающийся на инфраструктуру LLVM. Например, компилятор может использовать анализаторы исходного

кода и генераторы выполняемого кода из LLVM, но осуществлять окончательную компоновку с применением GCC и системных библиотек GNU. Такое значение слово «LLVM» имеет, например, в предложении: «Компиляция программ на C для платформы MIPS выполнялась с помощью LLVM».

- **Библиотеки LLVM:** Это часть кода инфраструктуры LLVM, допускающего многократное использование. Такое значение слово «LLVM» имеет, например, в предложении: «Для генерации выполняемого кода в моем проекте используется динамический компилятор LLVM».
- **Ядро LLVM:** Механизм оптимизации на уровне промежуточного языка и внутренние алгоритмы образуют ядро LLVM, с которого начинался проект. Такое значение слово «LLVM» имеет, например, в предложении: «LLVM и Clang – это два разных проекта».
- **LLVM IR:** Это – компилятор LLVM в промежуточное представление. Такое значение слово «LLVM» имеет, например, в предложении: «Я сконструировал анализатор исходного кода, выполняющий трансляцию программ на моем собственном языке в LLVM».

Чтобы понять суть проекта LLVM, необходимо познакомиться с наиболее важными частями инфраструктуры:

- **Анализатор исходного кода (frontend):** Компилятор, выполняющий этап трансляции программ на языках высокого уровня, таких как C, C++ и Objective-C, в промежуточное представление LLVM IR. Включает лексический, синтаксический (парсер) и семантический анализаторы, а также генератор кода LLVM IR. Проект Clang реализует все функции анализатора исходного кода, поддерживает возможность расширения подключаемыми модулями и позволяет использовать отдельный статический анализатор для более глубокого анализа кода. Дополнительные подробности приводятся в главе 4, «Анализатор исходного кода», в главе 9, «Статический анализатор Clang», и в главе 10, «Инструменты Clang и фреймворк LibTooling».
- **Промежуточное представление (IR):** Промежуточное представление LLVM IR имеет две формы: удобочитаемую текстовую форму и двоичную. Инструменты и библиотеки поддерживают интерфейсы для создания, ассемблирования и дисассемблирования IR. Оптимизатор LLVM также оперирует

промежуточным представлением IR, к которому применяется большая часть оптимизаций. Подробнее о промежуточном представлении IR рассказывается в главе 5, «Промежуточное представление LLVM».

- **Генератор выполняемого кода (backend):** Компилятор, преобразующий промежуточное представление LLVM IR программы в конкретный код на языке ассемблера для заданной аппаратной архитектуры или в двоичный объектный код. Генератор выполняемого кода отвечает за распределение регистров, преобразование циклов, локальные оптимизации и архитектурно-зависимые оптимизации/преобразования. Подробнее о генераторах выполняемого кода рассказывается в главе 6, «Генератор выполняемого кода».

На следующей диаграмме (рис. 3.1) показано место каждого компонента в проекте и общая организация инфраструктуры в конкретной конфигурации. Имейте в виду, что мы легко можем реорганизовать структуру, например, исключить компоновщик LLVM IR, если не собираемся заниматься исследованиями оптимизаций времени компоновки.

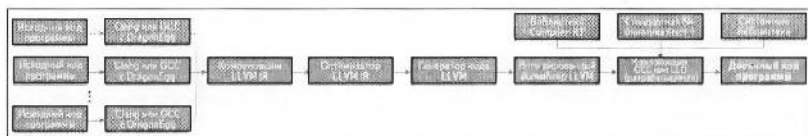


Рис. 3.1. Общая организация инфраструктуры в конкретной конфигурации

Все эти компоненты компилятора взаимодействуют друг с другом двумя способами:

- **Через структуры в памяти:** Взаимодействия этого вида протекают при посредничестве единого инструмента-супервизора, такого как Clang, который использует каждый компонент LLVM как библиотеку и опирается на структуры данных в памяти для передачи информации из одной стадии в другую.
- **Через файлы:** Взаимодействия этого вида протекают при посредничестве пользователя, который запускает отдельные инструменты, сохраняющие результаты работы того или иного компонента на диске, и получающие на входе указанные им файлы.

Как результат, инструменты высокого уровня, такие как Clang, могут включать в работу более мелкие инструменты, используя библиотеки, реализующие их функциональность. Это возможно, благодаря организации LLVM, основной упор в которой делается на максимально широкое использование библиотек. Не менее полезными оказываются и небольшие самостоятельные инструменты, использующие небольшое число библиотек, позволяя пользователям непосредственно взаимодействовать с конкретными компонентами LLVM посредством командной строки.

Например, взгляните на следующую диаграмму (рис. 3.2). На ней приводятся имена инструментов (выделены жирным шрифтом) и библиотек, которые они используют. На этой диаграмме видно, что инструмент `llc` (генератор выполняемого кода), использует библиотеку `libLLVMCodeGen`, реализующую часть его функциональных возможностей, а команда `opt`, которая вызывает оптимизатор LLVM IR, использует другую библиотеку – `libLLVMipa` – реализующую архитектурно-независимые межпроцедурные оптимизации. Также на диаграмме изображен более крупный инструмент `clang`, использующий обе библиотеки, замещающий инструменты `llc` и `opt` и предоставляющий более простой пользовательский интерфейс. То есть, любую задачу, выполняемую такими высокоуровневыми инструментами, можно разложить в цепочку вызовов низкоуровневых инструментов и получить тот же самый результат. В следующих разделах эта концепция рассматривается более подробно. В действительности Clang может выполнить всю цепочку компиляции, а не только действия, выполняемые инструментами `opt` и `llc`. Это объясняет, почему выполняемый файл Clang, скомпонованный с библиотеками статически, часто оказывается больше, чем сумма этих двух библиотек – он включает в себя всю экосистему LLVM.

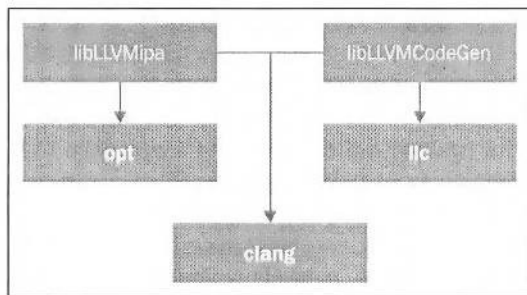


Рис. 3.2. Инструменты высокого и низкого уровня



Взаимодействие с драйвером компилятора

Драйвер компилятора можно сравнить с официантом в ресторане: он принимает ваш заказ, передает его повару и затем приносит готовые блюда, возможно, с какими-либо приправами. Драйвер отвечает за интеграцию всех необходимых библиотек и инструментов, чтобы обеспечить максимум удобств для пользователя и освободить его от необходимости вручную выполнять разные стадии компиляции, такие как парсинг, оптимизация, ассемблирование и компоновка. После передачи драйверу компилятора исходного кода программы, он создаст выполняемый файл. В LLVM и Clang функции драйвера компилятора выполняет инструмент `clang`.

Возьмем в качестве примера простую программу на языке C, `hello.c`:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Чтобы создать выполняемый файл из этой простой программы, достаточно выполнить следующую команду:

```
$ clang hello.c -o hello
```

Совет. Чтобы получить готовую к использованию версию LLVM, используйте инструкции из главы 1, «Сборка и установка LLVM».

Знакомые с компилятором GCC наверняка заметят поразительное сходство с соответствующей командой GCC. В действительности компилятор Clang проектировался так, чтобы обеспечить максимальную совместимость с GCC по флагам и структуре команд, чтобы LLVM можно было использовать взамен GCC. Clang для Windows имеет также версию команды `clang-cl.exe`, имитирующую интерфейс командной строки компилятора Visual Studio C++. Драйвер компилятора Clang неявно вызывает все остальные инструменты, от анализатора исходного кода до компоновщика.

Чтобы увидеть, какие инструменты вызываются драйвером и в какой последовательности, добавьте в команду аргумент `-###`:


```
$ clang -### hello.c -o hello
```

```
clang version 3.4 (tags/RELEASE_34/final)
```

```
Target: x86_64-apple-darwin11.4.2
```

```
Thread model: posix
```

```
"/bin/clang" -cc1 -triple x86_64-apple-macosx10.7.0 ... -main-file-name  
hello.c (...) /examples/hello/hello.o -x c hello.c
```

```
"/opt/local/bin/ld" (...) -o hello /examples/hello/hello.o (...)
```

В первую очередь драйвер компилятора Clang вызывает сам инструмент `clang` с параметром `-cc1`, отключающим режим драйвера компилятора и включающим режим простого компилятора. Дополнительно передается еще множество параметров, с настройками компиляции C/C++. Так как компоненты LLVM являются библиотеками, инструмент `clang -cc1` скомпонован с библиотеками генератора кода IR для целевой архитектуры и ассемблера. Поэтому после парсинга команда `clang -cc1` сама может вызывать другие библиотеки и управлять процессом компиляции в памяти, пока не будет получен объектный код. Далее драйвер Clang (не путайте с компилятором `clang -cc1`) вызывает компоновщика – внешний инструмент – который создаст выполняемый файл, как показано выше. В данном случае используется системный компоновщик, потому что компоновщик LLVM, `lld`, все еще находится на стадии разработки.

Имейте в виду, что взаимодействия через структуры в памяти протекают намного быстрее, чем через дисковые файлы, что делает компиляцию через файлы малопривлекательной. Это объясняет, почему Clang, анализатор исходного кода в LLVM и первый инструмент, получающий исходную программу, организует взаимодействия между остальными этапами компиляции через структуры в памяти, а не через промежуточные файлы.

Использование автономных инструментов

Тот же самый процесс компиляции, что был представлен выше, можно выполнить с использованием автономных инструментов LLVM, передавая вывод одного инструмента на ввод другого. Скорость компиляции при этом уменьшится, из-за необходимости сохранения

промежуточных результатов на диск, однако будет весьма поучительно увидеть всю последовательность действий. Кроме того, при таком подходе открывается возможность тонкой настройки параметров для промежуточных инструментов. Вот некоторые из этих инструментов:

- `opt`: выполняет оптимизацию программы на уровне промежуточного представления IR. На вход этого инструмента подается файл с биткодом LLVM (промежуточное представление LLVM IR в двоичной форме), а на выходе получается файл того же типа.
- `llc`: выполняет преобразование биткода LLVM в исходный текст на языке ассемблера для данной аппаратной архитектуры или в объектный код, с использованием соответствующего генератора выполняемого кода. Принимает аргументы, определяющие уровень оптимизации, с помощью которых можно исключить добавление отладочной информации в выходной файл и разрешить или запретить архитектурно-зависимые оптимизации.
- `llvm-mc`: выполняет трансляцию ассемблерного кода и может создавать объектные файлы в нескольких форматах, таких как ELF, MachO и PE. Может также дизассемблировать файлы, производя эквивалентный ассемблерный код и соответствующие внутренние конструкции LLVM.
- `lli`: реализует интерпретатор и JIT-компилятор для LLVM IR.
- `llvm-link`: выполняет компоновку нескольких файлов с биткодом LLVM bitcodes в один большой файл с тем же биткодом LLVM.
- `llvm-as`: преобразует файлы с промежуточным представлением LLVM IR в текстовом виде, которые называют сборками, LLVM, в файлы с биткодом LLVM.
- `llvm-dis`: декодирует биткод LLVM в сборки LLVM.

Рассмотрим простую программу на языке C, где используются функции из нескольких исходных файлов. Первый файл `main.c`:

```
#include <stdio.h>

int sum(int x, int y);

int main() {
    int r = sum(3, 4);
    printf("r = %d\n", r);
    return 0;
}
```

Второй файл `sum.c`:

```
int sum(int x, int y) {  
    return x+y;  
}
```

Скомпилировать эту программу можно командой:

```
$ clang main.c sum.c -o sum
```

Однако, тот же результат можно получить с помощью автономных инструментов. Сначала с помощью команды `clang` для каждого исходного файла создадим файл с биткодом LLVM:

```
$ clang -emit-llvm -c main.c -o main.bc  
$ clang -emit-llvm -c sum.c -o sum.bc
```

Флаг `-emit-llvm` сообщает инструменту `clang`, что он должен создать либо файл с биткодом LLVM, либо файл сборки LLVM, в зависимости от наличия флага `-c` или `-S`. В предыдущем примере флаг `-emit-llvm` используется в паре с флагом `-c`, поэтому `clang` создаст объектные файлы с биткодом LLVM. Применение комбинации флагов `-flto -c` даст тот же результат. Если вашей целью является получение сборки LLVM в удобочитаемом текстовом формате, используйте следующую пару команд:

```
$ clang -emit-llvm -S -c main.c -o main.ll  
$ clang -emit-llvm -S -c sum.c -o sum.ll
```

Примечание. Обратите внимание, что без флага `-emit-llvm` или `-flto`, флаг `-c` создаст объектный файл с машинным кодом, а флаг `-S` – файл с исходным кодом на языке ассемблера. Такое поведение инструмента совместимо с поведением GCC.

Расширения `.bc` и `.ll` используются для обозначения файлов с биткодом LLVM и файлов сборок, соответственно. Продолжить компиляцию дальше можно двумя путями:

- преобразовать биткод в объектные файлы и затем построить из них выполняемый файл программы, путем компоновки с помощью системного компоновщика (путь А на рис. 3.3):

```
$ llc -filetype=obj main.bc -o main.o  
  
$ llc -filetype=obj sum.bc -o sum.o  
  
$ clang main.o sum.o -o sum
```

- сначала скомпоновать два файла с биткодом LLVM в один общий файл, затем преобразовать его в объектный файл и из него

получить выполняемый файл программы вызовом системного компоновщика (путь **Б** на рис. 3.3):

```
$ llvm-link main.bc sum.bc -o sum.linked.bc
```

```
$ llc -filetype=obj sum.linked.bc -o sum.linked.o
```

```
$ clang sum.linked.o -o sum
```

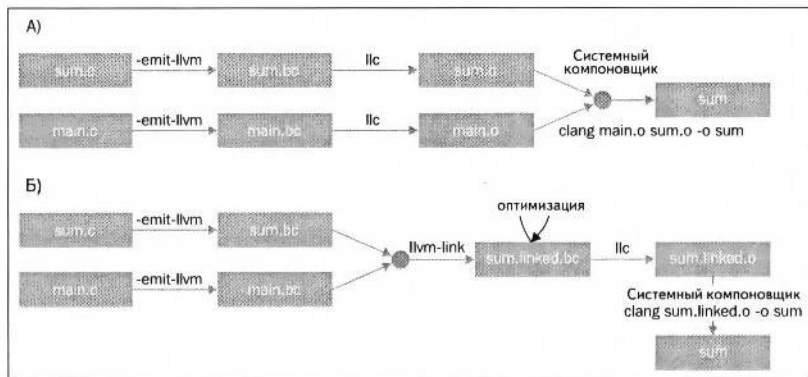


Рис. 3.3. Два пути продолжения компиляции

Параметр `-filetype=obj` указывает, что на выходе должен быть получен объектный файл, а не файл сборки. В этом примере мы вызываем системного компоновщика с помощью драйвера Clang, `clang`, однако его можно вызвать непосредственно, если вы знаете, какие параметры передать, чтобы скомпоновать программу с системными библиотеками.

Компоновка файлов с промежуточным представлением IR перед вызовом генератора выполняемого кода (`llc`) дает возможность применить дополнительные оптимизации к полученному промежуточному представлению IR вызовом инструмента `opt` (например, см. главу 5, «Промежуточное представление LLVM»). С другой стороны, инструмент `llc` может сгенерировать ассемблерный файл, который можно скомпилировать с помощью `llvm-mc`. Подробнее об этом интерфейсе мы расскажем в главе 6, «Генератор выполняемого кода».

Внутренняя организация LLVM

Чтобы разделить компилятор на несколько инструментов, архитектура LLVM предусматривает возможность взаимодействий между ком-

понентами на высоком уровне абстракции. Вследствие этого разные компоненты реализованы в виде отдельных библиотек, на языке C++, с использованием объектно-ориентированных парадигм и поддерживаемой расширяемой интерфейса, позволяющего легко интегрировать в конвейер компиляции преобразования и оптимизации.

Основные библиотеки LLVM

Логика LLVM и Clang организована в следующие библиотеки:

- `libLLVMCore`: Содержит всю логику, так или иначе связанную с LLVM IR: конструирование IR (представление данных, инструкции, базовые блоки и функции) и проверка IR. Также содержит реализацию диспетчера компиляции.
- `libLLVMAnalysis`: Содержит реализацию нескольких проходов анализа IR, таких как анализ псевдонимов, анализ зависимостей, свертка констант, сбор информации о циклах, анализ зависимости от памяти и упрощение инструкций.
- `libLLVMCodeGen`: Реализует генератор кода, независимого от целевой платформы, – низкоуровневая версия LLVM IR – анализ и преобразование.
- `libLLVMTarget`: Обеспечивает доступ к информации о целевой архитектуре посредством обобщенных абстракций. Эти высокоуровневые абстракции образуют мост между обобщенными алгоритмами генератора выполняемого кода, реализованными в `libLLVMCodeGen`, и архитектурно-зависимой логикой, зарезервированной для следующей библиотеки.
- `libLLVMX86CodeGen`: Включает информацию, необходимую для создания кода для архитектуры x86, а также проходы преобразования и анализа, составляющие генератор выполняемого кода x86. Обратите внимание, что для других аппаратных архитектур имеются свои, отдельные библиотеки, такие как `LLVMARMCodeGen` и `LLVMMipsCodeGen`, реализующие генераторы выполняемого кода для ARM и MIPS, соответственно.
- `libLLVMSupport`: Включает коллекцию утилит. Примерами алгоритмов, реализованных в этой библиотеке и используемых повсюду в инфраструктуре LLVM, могут служить: обработка ошибок, целых и вещественных чисел, парсинг аргументов командной строки, отладка, поддержка файлов и операции со строками.
- `libclang`: Реализует интерфейс для языка C, в противоположность C++, который является языком по умолчанию ре-

ализации LLVM, для доступа к большей части функций в Clang – получение диагностической информации, обход дерева синтаксического анализа AST, дополнение кода, отображение между курсорами и исходным кодом. Так как интерфейс C имеет более простую организацию, это упрощает его использование в проектах, написанных на других языках, таких как Python, хотя основное предназначение интерфейса для C заключается в том, чтобы обеспечить более высокую стабильность и служить опорой для внешних проектов. Он охватывает только подмножество интерфейса C++, используемого внутренними компонентами LLVM.

- `libclangDriver`: Содержит множество классов, используемых драйвером компилятора для анализа GCC-подобных параметров командной строки и преобразования их в адекватные параметры для внешних инструментов, выполняющих разные этапы компиляции. Может выбирать разные стратегии компиляции, в зависимости от целевой платформы.
- `libclangAnalysis`: Множество различных видов анализа уровня Clang. Реализует конструирование графа управляющей логики (CFG) и графа вызовов, проверку достижимости кода и безопасность строк формата, и многое другое.

В качестве примера использования этих библиотек для конструирования инструментов LLVM, на рис. 3.4 показаны зависимости инструмента `llc` от библиотек `libLLVMCodeGen`, `libLLVMTarget` и других, а также зависимости этих библиотек от третьих библиотек. Имейте в виду, что здесь представлен неполный список.

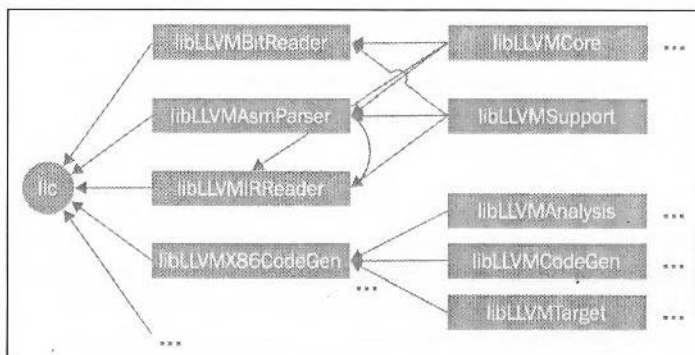


Рис. 3.4. Зависимости инструмента `llc`

Остальные библиотеки, не попавшие в этот начальный обзор, мы рассмотрим в следующих главах. Для версии 3.0, команда LLVM написала отличный документ, демонстрирующий зависимости между всеми библиотеками LLVM. Даже при том, что этот документ уже устарел, он содержит весьма интересный обзор организации библиотек. Прочитать его можно по адресу: <http://llvm.org/releases/3.0/docs/UsingLibraries.html>.

Приемы программирования на C++ в LLVM

Библиотеки и инструменты LLVM написаны на языке C++, чтобы получить дополнительные преимущества, которые дает объектно-ориентированная парадигма, и упростить взаимодействия между отдельными компонентами. Кроме того, в ходе разработки применялись наиболее эффективные приемы программирования на C++, с целью увеличить производительность, насколько это возможно.

Применение полиморфизма

Наследование и полиморфизм широко используются в генераторах выполняемого кода, помогая вынести реализацию обобщенных алгоритмов в базовые классы. При таком подходе каждый конкретный генератор выполняемого кода может сосредоточиться на реализации индивидуальных особенностей, переопределяя только необходимые методы суперкласса. Библиотека `LibLLVMCodeGen` содержит реализацию обобщенных алгоритмов, а `LibLLVMTarget` – интерфейсы и абстракции отдельных архитектур. Следующий фрагмент кода (из `llvm/lib/Target/Mips/MipsTargetMachine.h`) показывает, как объявляется класс поддержки архитектуры MIPS, наследующий суперкласс `LLVMTargetMachine`, иллюстрируя данную концепцию. Этот код является частью библиотеки `LLVMMipsCodeGen`:

```
class MipsTargetMachine : public LLVMTargetMachine {
    MipsSubtarget Subtarget;
    const DataLayout DL;
    ...
}
```

Чтобы причины выбора языка C++ разработчиками стали более очевидными, мы покажем еще один пример генератора выполняемого кода, в котором механизм распределения регистров, типичный для всех генераторов кода, должен знать, какие регистры уже заняты и не могут выделяться повторно. Эта информация во многом зависит от целевой аппаратной архитектуры и не может быть представлена в

обобщенном суперклассе. Данная задача решается с помощью метода `MachineRegisterInfo::getReservedRegs()`, который должен переопределяться для каждой целевой архитектуры. Следующий фрагмент (из `llvm/lib/Target/Sparc/SparcRegisterInfo.cpp`) показывает, как генератор выполняемого кода для архитектуры SPARC переопределяет этот метод:

```
BitVector SparcRegisterInfo::getReservedRegs(...) const {
    BitVector Reserved(getNumRegs());
    Reserved.set(SP::G1);
    Reserved.set(SP::G2);
    ...
}
```

Данный генератор кода для SPARC конструирует битовый вектор, отбирая регистры, которые не могут использоваться для обычного распределения.

Шаблоны C++ в LLVM

В LLVM широко используются шаблоны C++, но без злоупотреблений, которые увеличивают и без того большое время компиляции проектов на C++. Везде, где возможно, используется специализация шаблонов, чтобы обеспечить высокую скорость работы реализации. В качестве примера использования шаблонов в LLVM, рассмотрим функцию, проверяющую, укладывается ли целое число, переданное в параметре, в заданное число битов, которое является параметром шаблона (пример взят из `llvm/include/llvm/Support/MathExtras.h`):

```
template<unsigned N>
inline bool isInt(int64_t x) {
    return N >= 64 ||
        (!(INT64_C(1) << (N-1)) <= x && x < (INT64_C(1) << (N-1)));
}
```

Обратите внимание, как шаблон обрабатывает все значения числа битов, `n`. В нем используется предварительное сравнение, обеспечивающее возврат значения `true`, если заданное число битов больше 64. В противном случае вычисляются два выражения, возвращающие нижнюю и верхнюю границы диапазона чисел заданного размера, и указанное значение `x` сравнивается с этими границами. Сравните этот код со следующей специализацией шаблона, которая применяется для ускорения в случаях, когда заданное число битов равно 8:

```
llvm/include/llvm/Support/MathExtras.h:
template<>
```



```
inline bool isInt<8>(int64_t x) {
    return static_cast<int8_t>(x) == x;
}
```

Этот код уменьшает число сравнений с трех до одного, тем самым оправдывая специализацию.

Эффективные приемы программирования на C++ в LLVM

Всем нам свойственно ошибаться, и все мы непреднамеренно допускаем ошибки в программах. Но все мы по-разному относимся к предупреждению таких ошибок. Философия LLVM требует использовать везде, где это возможно, механизм утверждений (assertions), реализованный в libLLVMsupport. Имейте в виду, что отладка компилятора дается особенно тяжело, потому что продуктом компиляции является другая программа. Поэтому, возможность раннего определения ошибочного поведения, до получения объемного результата компиляции, в котором сложно будет разобраться, поможет сэкономить массу времени. Например, взгляните на реализацию генератора выполняемого кода для архитектуры ARM, которая изменяет пулы констант, перераспределяя их между несколькими пулами меньшего размера внутри функции. Эта стратегия широко используется в программах для архитектуры ARM с целью избежать загрузки больших констант в области памяти, отстоящие слишком далеко от инструкций, использующих. Этот код находится в `llvm/lib/Target/ARM/ARMConstantIslandPass.cpp` и ниже показана выдержка из него:

```
const DataLayout &TD = *MF->getTarget().getDataLayout();
for (unsigned i = 0, e = CPs.size(); i != e; ++i) {
    unsigned Size = TD.getTypeAllocSize(CPs[i].getType());
    assert(Size >= 4 && "Too small constant pool entry");
    unsigned Align = CPs[i].getAlignment();
    assert(isPowerOf2_32(Align) && "Invalid alignment");
    // Проверить, все ли элементы пула констант кратны величине выравнивания.
    // Иначе их надо увеличить, чтобы сохранить выравнивание инструкций.
    assert((Size % Align) == 0 && "CP Entry not multiple of 4 bytes!");
}
```

В этом фрагменте выполняется обход структуры данных, представляющей пул констант ARM, и программист ожидает, что каждое поле этого объекта будет соответствовать определенным ограничениям. Обратите внимание, как программист сохраняет семантику данных под своим контролем, используя вызовы `assert`. Если будет обнаружено отклонение от ожидаемого, его программа немедленно пре-



кратит выполнение и выведет сообщение об ошибке. Программист использует идиому объединения булева выражения с `&&` "текстом сообщения об ошибке!", которое не влияет на вычисление булева выражения, но дает краткое пояснение, описывающее причины неудачи. Подобные проверки ухудшают производительность, поэтому они полностью устраняются при компиляции окончательной версии проекта LLVM, когда такие проверки автоматически запрещаются.

Также часто в коде LLVM можно встретить использование «интеллектуальных указателей» (smart pointers). Они поддерживают автоматическое освобождение памяти, как только имя оказывается вне области видимости, и используются в LLVM, например, для хранения информации о целевой архитектуре и модулях. В прошлом в LLVM использовался специальный класс интеллектуальных указателей `OwningPtr`, который определен в `llvm/include/llvm/ADT/OwningPtr.h`. Начиная с версии LLVM 3.5, этот класс перешел в разряд нерекомендуемых к использованию и вместо него стали использоваться встроенные интеллектуальные указатели `std::unique_ptr()`, введенные стандартом C++11.

Если вам интересно увидеть полный перечень эффективных приемов программирования на C++, используемых в проекте LLVM, посетите страницу <http://llvm.org/docs/CodingStandards.html>. С ней стоит ознакомиться всем программистам на C++.

Легковесные ссылки на строки в LLVM

В состав проекта LLVM входит обширная библиотека структур данных и универсальных алгоритмов, и строки занимают в ней особое место. Они часто вызывают жаркие дискуссии в среде программистов на C++: когда лучше использовать простой тип `char*`, а когда класс `string` из стандартной библиотеки C++? В проекте LLVM ссылки на строки используются очень широко, например, для передачи имен модулей, функций и значений. В некоторых случаях допускается присутствие в строках нулевых символов, из-за чего делается невозможной передача строковых констант, как указателей типа `const char*`, потому что в C нулевой символ считается завершающим символом строки. С другой стороны, использование `const std::string&` приводит к необходимости выделения дополнительной памяти в куче, потому что экземпляр класса `string` должен где-то хранить свой буфер для символов. Рассмотрим следующий пример:

```
bool hasComma (const std::string &a) {  
    // код
```

```

}

void myfunc() {
    char buffer [40];
    // код создает строку в собственном буфере
    hasComma(buffer); // компилятор C++ вынужден создать новый
                    // объект строки, дублирующий буфер
    hasComma("hello, world!"); // аналогично
}

```

Обратите внимание, что при каждой попытке создать строку в отдельном буфере, в куче выделяется память для объекта копии этой строки, который имеет собственный буфер. В первом случае строка размещается на стеке, тогда как во втором случае строка представляет собой глобальную константу. Чего не хватает в C++ для подобных случаев, так это простого класса, который помог бы избежать ненужного выделения памяти, когда единственное, что нужно – это простая ссылка на строку. Даже если работать только со строковыми объектами `string`, чтобы исключить ненужные операции с выделением памяти в куче, использование ссылки на объект `string` предполагает два уровня косвенности. Поскольку класс `string` уже использует внутренний указатель на данные, передача указателя на объект `string` предполагает разыменование двух указателей для доступа к фактическим данным.

Повысить эффективность выполнения в таких ситуациях позволяет класс, имеющийся в LLVM, поддерживающий работу со ссылками на строки: `StringRef`. Это легковесный класс, экземпляры которого можно передавать по значению, точно так же как указатели типа `const char*`, но они дополнительно хранят размеры строк, что позволяет иметь пулевые символы в строках. Однако, в отличие от объектов `string`, они не имеют собственных буферов и поэтому никогда не требуют выделения памяти в куче, храня лишь указатели на строки, хранящиеся где-то еще. Использование этой концепции можно также наблюдать в других проектах на C++. Например, ту же идею в Chromium реализует класс `StringPiece`.

В LLVM имеется еще один класс для работы со строками. Собрать строку из нескольких других строк путем конкатенации, в LLVM можно с помощью класса `Twine`. Он откладывает фактическую конкатенацию, храня только ссылки на строки, составляющие конечный результат. Этот класс был создан еще до появления стандарта C++11, когда конкатенация строк была весьма дорогостоящей операцией.

Если вам интересно поближе познакомиться с другими универсальными классами, которые проект LLVM предоставляет в помощь программистам, обязательно сохраните в своих закладках ссылку на документ «LLVM Programmer's Manual», где обсуждаются все универсальные структуры данных LLVM, которые могут пригодиться в любых программах. Этот документ находится по адресу: <http://llvm.org/docs/ProgrammersManual.html>.

Демонстрация расширяемого интерфейса проходов

Под «проходом» в данном случае понимается анализ преобразования или оптимизация. LLVM API позволяет регистрировать собственные проходы на разных этапах компиляции, что является ценнейшей особенностью проекта LLVM. Регистрация проходов, их планирование и объявление зависимостей между проходами выполняются с помощью диспетчера проходов. Соответственно, экземпляры класса `PassManager` доступны на всех стадиях компиляции.

Например, цели могут предусматривать собственные оптимизации в разных точках процесса создания кода, например, до и после распределения регистров или перед выводом сборки. Чтобы продемонстрировать эту возможность, мы покажем вам пример, когда при определенных условиях цель X86 регистрирует пару проходов, выполняемых перед выводом сборки (фрагмент взят из `lib/Target/X86/X86TargetMachine.cpp`):

```
bool X86PassConfig::addPreEmitPass() {
    ...
    if (getOptLevel() != CodeGenOpt::None && getX86Subtarget().hasSSE2()) {
        addPass(createExecutionDependencyFixPass(&X86::VR128RegClass));
        ...
    }

    if (getOptLevel() != CodeGenOpt::None &&
        getX86Subtarget().padShortFunctions()) {
        addPass(createX86PadShortFunctions());
    }
    ...
}
```

Обратите внимание, как генератор выполняемого кода (backend) определяет необходимость регистрации дополнительного прохода, проверяя информацию о целевой архитектуре. Перед добавлением первого прохода проверяется поддержка мультимедийных расшире-

ний SSE2. Перед добавлением второго прохода проверяется наличие требований к выравниванию.

В части А на рис. 3.5 показано, как выполняется добавление проходов в инструменте `opt`, а часть Б иллюстрирует несколько точек регистрации в процедуре создания выполняемого кода, где могут быть зарегистрированы дополнительные оптимизации. Обратите внимание, что точки регистрации разбросаны по разным этапам создания выполняемого кода. Данная диаграмма пригодится вам, когда вы будете писать свой первый проход и решать, когда он должен выполняться. Интерфейс `PassManager` подробно описывается в главе 5, «Промежуточное представление LLVM».

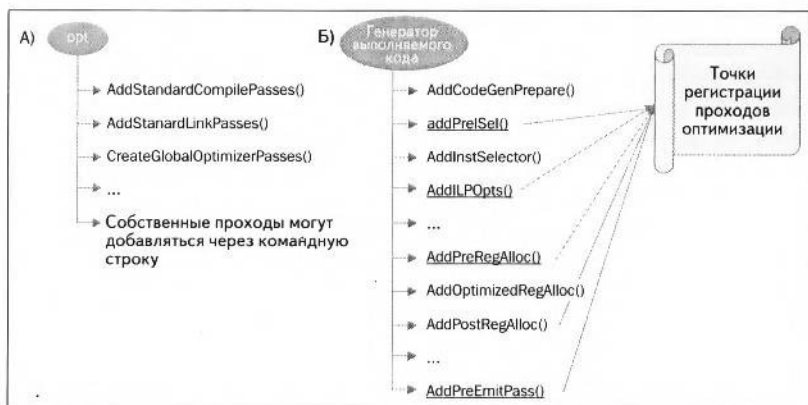


Рис. 3.5. Добавление проходов в инструменте `opt` и точки регистрации в процедуре создания выполняемого кода

Реализация первого собственного проекта LLVM

В этом разделе мы покажем, как написать первый собственный проект, использующий библиотеки LLVM. В предыдущих разделах мы показали, как пользоваться инструментами LLVM, чтобы получить файл с промежуточным представлением программы – файл биткода. Теперь мы напишем программу, которая будет читать этот файл и выводить имена функций, объявленных в нем, и номера их основных блоков, чтобы показать, насколько просто пользоваться библиотеками LLVM.

Makefile

Компоновка с библиотеками LLVM требует ввода длинных команд, которые сложно писать вручную, без помощи системы сборки. Мы покажем вам файл сборки Makefile, за основу которого взят один из таких файлов в проекте DragonEgg, и объясним каждый его раздел. Если просто скопировать код из книги, вы потеряете символы табуляции; помните, что символы табуляции играют важную роль в определениях правил в файлах Makefile. То есть, вам придется вручную вставить их:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags)
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
```

В этом первом разделе определяются переменные, которые будут использоваться для передачи флагов компилятору. Первая переменная определяет местоположение программы `llvm-config`. В данном случае здесь должен быть путь к этой программе в вашей системе. Программа `llvm-config` – это инструмент LLVM, который выводит массу полезной информации, которая может пригодиться при компоновке внешних проектов с библиотеками LLVM.

Далее, определяя множество флагов для компилятора C++, например, мы предлагаем утилите `make` получить эти флаги, выполнив команду `llvm-config --cxxflags`. Таким способом достигается совместимость среды компиляции исходных текстов нашего проекта со средой компиляции исходных текстов LLVM. Последняя переменная определяет множество флагов для переадресации препроцессору компилятора.

```
HELLO=helloworld
HELLO_OBJECTS=hello.o

default: $(HELLO)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(HELLO) : $(HELLO_OBJECTS)
```

```
@echo Linking $@
$(QUIET)$ (CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ `$(LLVM_CONFIG)
--libs bitreader core support`
```

В этом втором разделе определяются правила сборки. Первым следует правило по умолчанию `default`, описывающее сборку выполняемого файла `hello-world`. Второе правило – универсальное, осуществляющее компиляцию всех файлов с исходными текстами на C++ в объектные файлы. Мы указали здесь флаги препроцессора и флаги компилятора C++. Здесь также использована переменная `$(QUIET)`, чтобы предотвратить вывод на экран полных команд компиляции, но, если вы пожелаете видеть подробный вывод в процессе сборки, определите переменную окружения `VERBOSE` перед вызовом GNU Make.

Последнее правило описывает связывание объектных файлов – в данном случае получится единственный такой файл – в общий выполняемый файл программы, скомпонованный с библиотеками LLVM. Операция компоновки выполняется компоновщиком, но он может принимать некоторые флаги C++. Поэтому мы предусмотрели передачу компоновщику не только его флагов, но и флагов компилятора C++. Команда вызова компоновщика завершается конструкцией ``command``, которая предписывает командной оболочке заменить эту часть команды выводом команды ``command``. В данном случае командой `command` является команда `llvm-config --libs bitreader core support`. Флаг `--libs` требует от `llvm-config` вывести список флагов компоновки с указанными библиотеками LLVM. В данном случае мы перечислили библиотеки `libLLVMBitReader`, `libLLVMCore` и `libLLVMSupport`.

Список флагов возвращается программой `llvm-config` в виде последовательности параметров `-l`, например: `-lLLVMCore -lLLVMSupport`. Однако имейте в виду, что порядок следования параметров при вызове компоновщика имеет значение и первыми должны следовать библиотеки, зависящие от других. Например, библиотека `libLLVMCore` использует функции из библиотеки `libLLVMSupport`, поэтому при вызове компоновщика они должны указываться в порядке: `-lLLVMCore -lLLVMSupport`.

Порядок имеет значение потому, что библиотека – это коллекция объектных файлов, и когда проект компоуется с библиотекой, компоновщик выбирает только те объектные файлы для добавления в сборку, которые содержат символы, не найденные в прежде обработанных объектных файлах. То есть, если библиотека в последнем



парамetre использует символ из библиотеки, которая уже была обработана, большинство компоновщиков (включая GNU ld) не вернется назад, чтобы включить, возможно, недостающий объектный файл в сборку, из-за чего возникает ошибка компоновки.

Чтобы заставить компоновщик многократно возвращаться назад, с целью разрешить все неизвестные символы во всех объектных файлах, используйте флаги `--start-group` и `--end-group` в начале и в конце списка библиотек, но имейте в виду, что это может замедлить компоновку программы. Чтобы избежать лишней головной боли, вручную определяя порядок следования библиотек при сборке разветвленного дерева зависимостей, можно просто воспользоваться командой `llvm-config --libs` и позволить ей сделать все необходимое за вас.

В последнем разделе файла `Makefile` определяется правило `clean`, управляющее удалением всех файлов, сгенерированных компилятором, чтобы дать возможность начать сборку «с нуля»:

```
clean::
    $(QUIET) rm -f $(HELLO) $(HELLO_OBJECTS)
```

Реализация

Ниже представлен весь код, реализующий наш проход. Он получился относительно коротким, потому что в значительной степени опирается на инфраструктуру проходов LLVM, которая делает большую часть работы за нас.

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/raw_os_ostream.h"
#include "llvm/Support/system_error.h"
#include <iostream>

using namespace llvm;

static cl::opt<std::string> FileName(cl::Positional,
                                     cl::desc("Bitcode file"), cl::Required);
int main(int argc, char** argv) {
    cl::ParseCommandLineOptions(argc, argv, "LLVM hello world\n");
    LLVMContext context;
    std::string error;
    OwningPtr<MemoryBuffer> mb;
    MemoryBuffer::getFile(FileName, mb);
```



```

Module *m = ParseBitcodeFile(mb.get(), context, &error);
if (m == 0) {
    std::cerr << "Error reading bitcode: " << error << std::endl;
    return -1;
}
raw_os_ostream O(std::cout);

for (Module::const_iterator i = m->getFunctionList().begin(),
     e = m->getFunctionList().end(); i != e; ++i) {
    if (!i->isDeclaration()) {
        O << i->getName() << " has " << i->size() << " basic block(s).\n";
    }
}
return 0;
}

```

Наша программа использует инструменты LLVM из пространства имен `cl` (имя `cl` происходит от англ. «command line» – командная строка) для реализации интерфейса командной строки. Мы просто вызываем функцию `ParseCommandLineOptions` и объявляем глобальную переменную типа `cl::opt<string>`, чтобы показать, что наша программа принимает единственный параметр строкового типа, через который передается имя файла с биткодом.

Далее создается объект `LLVMContext` для хранения всех данных, принадлежащих компилятору LLVM, что обеспечивает безопасность в многопоточной среде. Класс `MemoryBuffer` определяет интерфейс доступа только для чтения к блоку памяти. Он будет использоваться функцией `ParseBitcodeFile` для чтения содержимого входного файла и парсинга его содержимого в формате LLVM IR. Выполнив проверки на ошибки и убедившись, что все в порядке, начинаем выполнять итерации по всем функциям модуля в этом файле. Модуль LLVM – это свособразная единица трансляции, она содержит все, что имеется в файле биткода, находится на вершине иерархии LLVM, ниже в которой следуют функции, затем базовые блоки и, наконец, инструкции. Если функция только объявлена, но не реализована, мы отбрасываем ее, поскольку нам требуется проверить реализацию функций. Обнаружив определение функции, мы выводим ее имя и число базовых блоков, составляющих ее.

Скомпилируйте эту программу и запустите ее с флагом `-help`, чтобы проверить работу интерфейса командной строки, реализованного в LLVM. Далее, найдите файл с исходным кодом на языке C или C++, который вам хотелось бы преобразовать в промежуточное представление LLVM IR, выполните преобразование и проанализируйте только что созданной программой:

```
$ clang -c -emit-llvm mysource.c -o mysource.bc  
$ helloworld mysource.bc
```

За дополнительными сведениями о том, какую еще информацию можно извлечь из функций, обращайтесь к документации LLVM с описанием класса `llvm::Function`, доступной по адресу: http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html. В качестве упражнения, попробуйте расширить этот пример, добавив вывод списка аргументов для каждой функции.

Общие советы по навигации в исходных текстах LLVM

Прежде чем продолжить знакомство с реализацией LLVM, необходимо осветить некоторые моменты, в основном для программистов, которые только вступают в мир открытого программного обеспечения. Если бы вы работали с закрытым проектом внутри компании, вы наверняка получили бы поддержку от программистов, давно работающих над проектом и имеющих более полное представление об организационных решениях, которые вам могут показаться странными на первый взгляд. Если вы столкнетесь с проблемой, автор компонента наверняка разъяснит вам ее причины при личной встрече. Эффективность подобных разъяснений часто бывает очень высока, потому что ваш визави имеет возможность наблюдать выражение вашего лица, отмечать моменты, которые вы не понимаете и изменять ход беседы, чтобы объяснить суть на более понятных для вас примерах.

Однако, когда работа выполняется удаленно, что характерно для большинства открытых проектов, физический контакт отсутствует и остается только вербальный контакт. Поэтому в открытых проектах большое внимание уделяется четкой и ясной документации. С другой стороны, документация может не содержать описание на хорошем литературном языке, объясняющее все организационные решения. По большей части документацией является сам код, и в этом смысле существует серьезный побудительный мотив писать как можно более ясный код, который помог бы другим понять, что собственно происходит.

Читайте код как документацию

Даже при том, что для большей части LLVM имеется отличная документация и мы на протяжении всей книги отсылаем и будем отсылать вас к ней, наша конечная цель состоит в том, чтобы подготовить вас к непосредственному чтению кода, потому что это позволит вам глубже

вникнуть в инфраструктуру LLVM. Мы познакомим вас с базовыми понятиями, которые помогут вам понять, как действует LLVM, и, обладая этими знаниями, вы обретете радость понимания кода LLVM, без необходимости читать документацию, и даже сможете изучить многие компоненты LLVM, для которых вообще нет никакой документации. Несмотря на все сложности, начав делать это, вы будете все глубже проникать в суть проекта и обретать все более твердую уверенность в своих знаниях. Перед тем, как вы поймете это, вы станете опытным программистом, с глубоким пониманием внутреннего устройства LLVM и сможете оказывать помощь другим.

Обращайтесь за помощью к сообществу

Существуют списки рассылки, которые помогут вам ощутить себя не одиноками. Вопросы, касающиеся анализатора исходного кода Clang, можно задавать в списке рассылки `cfe-dev`, для вопросов о ядре LLVM предназначен список рассылки `llvmdev`. Уделите немного времени, подпишитесь на эти рассылки по следующим адресам:

- Clang Front End Developer List (<http://lists.cs.uiuc.edu/mailman/listinfo/cfe-dev>);
- LLVM core Developer List (<http://lists.cs.uiuc.edu/mailman/listinfo/llvmdev>).

Многие разработчики, работающие в проекте, в свое время могли попытаться реализовать что-то, что нужно вам. Поэтому, высока вероятность, что ваша проблема уже была кем-то решена.

Но, прежде чем просить помощи, постарайтесь напрячь свой мозг и попытаться выяснить интересующий вас вопрос чтением исходного кода. Посмотрите, как высоко вы можете взлететь без посторонней помощи и старайтесь сами накапливать свои знания. Если вы столкнетесь с чем-то, что покажется вам неразрешимой проблемой, напишите сообщение в список рассылки, подробно описав, какие шаги вы предприняли, прежде чем обращаться за помощью. Следуя этим рекомендациям, вы получаете намного больше шансов получить исчерпывающие ответы.

Знакомьтесь с обновлениями – читайте журнал изменений в SVN как документацию

Проект LLVM постоянно меняется и нередко можно обнаружить, что после очередного обновления версии LLVM часть вашей программы,

взаимодействующая с библиотеками LLVM, перестала работать. Прежде чем вновь погружаться в чтение кода, чтобы выяснить причины, ознакомьтесь с обзором изменений.

Чтобы увидеть, как это применяется на практике, возьмем в качестве примера обновление Clang с версии 3.4 до версии 3.5. Допустим, что вы написали свой статический анализатор, который создает экземпляр класса BugType:

```
BugType *bugType = new BugType("This is a bug name",
                               "This is a bug category name");
```

Этот объект используется вашими функциями проверки (подробнее об этом рассказывается в главе 9, «Статический анализатор Clang») для выявления ошибок определенных видов. После обновления версий LLVM и Clang, при попытке скомпилировать свой код, вы получили следующий результат:

```
error: no matching constructor for initialization of
      'clang::ento::BugType'
BugType *bugType = new BugType("This is a bug name",
                               ^~~~~~
```

Эта ошибка произошла из-за того, что изменился конструктор класса BugType. Если вам сложно понять, как адаптировать свой код, чтобы он соответствовал новой версии, загляните в журнал изменений, который является важнейшим документом, отмечающим все изменения кода за определенный период. К счастью для любого открытого проекта, использующего систему управления версиями, легко можно обратиться к серверу репозитория и получить все изменения для конкретного файла. В случае с проектом LLVM это можно сделать даже с помощью браузера, на странице ViewVC: <http://llvm.org/viewvc>.

В данном примере нас интересуют изменения в заголовочном файле, где определяется конструктор класса. Заглядываем в дерево с исходными текстами LLVM и находим его в каталоге include/clang/StaticAnalyzer/Core/BugReporter/BugType.h.

Совет. Если вы пользуетесь текстовым редактором, проверьте, есть ли возможность внедрить в него инструмент, который упростит навигацию по исходным текстам LLVM. Например, потратьте немного времени, чтобы выяснить, поддерживает ли ваш редактор STAGS. Используя этот инструмент вы быстро будете находить любые файлы в дереве каталогов с исходными текстами LLVM, где определяются интересующие вас классы. Если вы упрямы и ни в какую не желаете пользоваться STAGS или каким-то другим инстру-

ментом, помогающим в навигации по большим проектам C/C++ (таким как *Visual Studio IntelliSense* или *Xcode*), вы всегда можете воспользоваться командами, такими как `grep -re "keyword" *`, выполняя их в корневом каталоге проекта, чтобы получить список файлов, содержащих искомое слово `keyword`. Тщательно выбирая слова для поиска, вы легко сможете находить файлы с определениями.

Чтобы увидеть сообщения сопровождающие изменения в данном конкретном заголовочном файле, можно открыть в браузере страницу <http://llvm.org/viewvc/llvm-project/cfe/trunk/include/clang/StaticAnalyzer/Core/BugReporter/BugType.h?view=log>, которая отобразит журнал изменений в браузере. Теперь можно увидеть, что случилось три месяца тому назад (то есть, за три месяца до написания этих строк):

```
Revision 201186 - (view) (download) (annotate) - [select for diffs]
Modified Tue Feb 11 15:49:21 2014 CST (3 months, 1 week ago) by alexfh
File length: 2618 byte(s)
Diff to previous 198686 (colored)
```

Expose the name of the checker producing each diagnostic message.

Summary: In `clang-tidy` we'd like to know the name of the checker producing each diagnostic message. `PathDiagnostic` has `BugType` and `Category` fields, which are both arbitrary human-readable strings, but we need to know the exact name of the checker in the form that can be used in the `CheckersControlList` option to enable/disable the specific checker. This patch adds the `CheckName` field to the `CheckerBase` class, and sets it in the `CheckerManager::registerChecker()` method, which gets them from the `CheckerRegistry`. Checkers that implement multiple checks have to store the names of each check in the respective `registerXXXChecker` method. **Reviewers:** `jordan_rose`, `krememek` **Reviewed By:** `jordan_rose` **CC:** `cfecommits` **Differential Revision:** <http://llvm-reviews.chandlerc.com/D2557>

Сообщение, сопровождающее изменение, достаточно подробно объясняет причины, побудившие изменить конструктор `BugType`: прежде объекты этого типа создавались с двумя строками, которых было недостаточно, чтобы определить, какой именно объект, выполняющий проверку, нашел ошибку. Поэтому теперь при создании экземпляра класса `BugType` конструктору должен передаваться экземпляр объекта, выполняющего проверку, чтобы потом проще было узнать, какой объект какую ошибку нашел.

Теперь изменим нашу программу в соответствии с изменением интерфейса. Предполагается, что данный код выполняется как часть функции-члена класса `Checker`, что является обычным делом

для объектов, выполняющих статический анализ. Поэтому объект Checker доступен по ссылке [this](#):

```
BugType *bugType = new BugType(this, "This is a bug name",  
                                "This is a bug category name");
```

Заключительные замечания

Когда вы слышите, что проект LLVM хорошо документирован, не ожидайте найти точное и полное описание всего кода. Под этими словами подразумевается, что если вы сможете читать код, интерфейсы, комментарии и сообщения, сопровождающие изменения в репозитории, вы сможете углублять свое понимание проекта LLVM и идти в ногу с последними изменениями. Не забывайте практиковаться в изучении исходного кода и исследовать работу разных механизмов, а это означает, что вам нужно держать готовым к исследованиям свой инструмент STAGS!

В заключение

В этой главе мы познакомили вас с историей организационных решений, принятых в проекте LLVM и дали краткий обзор наиболее важных из них. Мы также показали вам, что существует два разных способа использования компонентов LLVM. Первый – через драйвер компилятора, который является высокоуровневым инструментом, позволяющим выполнить всю процедуру компиляции одной командой. Второй – через автономные инструменты LLVM. Помимо сохранения промежуточных результатов на диске, что замедляет компиляцию, инструменты позволяют взаимодействовать с определенными фрагментами в библиотеках LLVM через командную строку, обеспечивая более точное управление процессом компиляции. Они являются отличным средством изучения особенностей работы LLVM. Мы также познакомили вас с некоторыми приемами программирования на C++, которые используются в LLVM, и объяснили, как исследовать документацию к LLVM и пользоваться помощью сообщества.

В следующей главе мы подробно рассмотрим реализацию анализатора исходного кода Clang и его библиотек.

ГЛАВА 4.

Анализатор исходного кода

Анализатор исходного кода преобразует исходный код в промежуточное представление перед созданием выполняемого кода. Так как языки программирования имеют разный синтаксис и семантику, каждый анализатор исходного кода обычно обрабатывает только один язык или группу похожих языков. Clang, например, обрабатывает исходный код на языках C, C++ и objective-C. В этой главе рассматриваются следующие темы:

- ♦ как компоновать программы с библиотеками Clang и как использовать `libclang`;
- ♦ этапы диагностики и анализа Clang;
- ♦ лексический, синтаксический и семантический анализ на примере использования `libclang`;
- ♦ как написать упрощенный драйвер компилятора, использующий библиотеки C++ Clang.

Введение в Clang

Проект Clang считается официальным анализатором исходного кода в LLVM для языков C, C++ и Objective-C. Официальный веб-сайт проекта Clang находится по адресу: <http://clang.llvm.org>. А настройка, сборка и установка Clang рассматривались в главе 1, «Сборка и установка LLVM».

Подобно названию LLVM, имеющему несколько значений, название Clang также может использоваться в трех разных смыслах:

1. Анализатор исходного кода (реализован как комплекс библиотек Clang).
2. Драйвер компилятора (реализован как команда `clang` и библиотека Clang Driver).
3. Фактический компилятор (реализован как команда `clang -cc1`). Компилятор `clang -cc1` реализован не только с при-

менением библиотек Clang, он также широко использует библиотеки LLVM, реализующие генераторы промежуточного и выполняемого кода, и интегрированный ассемблер.

В этой главе мы сосредоточимся на знакомстве с библиотеками Clang и анализаторе исходного кода на языках из семейства C для LLVM.

Исследование особенностей работы драйвера и компилятора мы начнем с анализа командной строки, вызывающей драйвер компилятора clang:

```
$ clang hello.c -o hello
```

После парсинга аргументов командной строки драйвер Clang вызывает внутренний компилятор, порождая еще один процесс вызовом еще одного экземпляра самого себя с параметром `-cc1`. Передав флаг `-Xclang <option>` драйверу компилятора, можно передать дополнительные параметры внутреннему компилятору, который, в отличие от драйвера, не стремится к совместимости с интерфейсом командной строки GCC. Например, инструмент clang `-cc1` имеет специальный параметр для вывода абстрактного синтаксического дерева Clang (Abstract Syntax Tree, AST):

```
$ clang -Xclang -ast-dump hello.c
```

Компилятор clang `-cc1` можно также вызвать непосредственно, минуя драйвер:

```
$ clang -cc1 -ast-dump hello.c
```

Но помните, что одной из задач драйвера компилятора является подготовка всех необходимых параметров и передача их компилятору. Чтобы увидеть все параметры, которые передаются компилятору clang `-cc1`, можно вызвать драйвер с флагом `-###`. Например, вызывая clang `-cc1` вручную, ему необходимо передать все пути к системным заголовочным файлам с помощью флага `-I`.

Работа анализатора исходного кода

Важным аспектом (и источником путаницы) инструмента clang `-cc1` является то обстоятельство, что он реализует не только анализатор исходного кода, но и весь цикл компиляции. То есть, использует все необходимые библиотеки и компоненты LLVM, чтобы выполнить всю процедуру компиляции до самого конца. То есть, он реализует *практически* законченный компилятор. Обычно, для архитектуры

x86, clang -ccl прекращает компиляцию после создания объектных файлов, потому что компоновщик LLVM все еще находится на стадии экспериментальной разработки. В этот момент он возвращает управление драйверу, который в свою очередь вызывает внешний инструмент для компоновки проекта. Флаг -### выводит список программ, вызываемых драйвером Clang:

```
$ clang hello.c -###
```

```
clang version 3.4 (tags/RELEASE_34/final 211335)
```

```
Target: i386-pc-linux-gnu
```

```
Thread model: posix
```

```
"clang" "-ccl" (...parameters) "hello.c" "-o" "/tmp/hello-dddafc1.o"
"/usr/bin/ld" (...parameters) "/tmp/hello-dddafc1.o" "-o" "hello"
```

Здесь мы опустили полный список параметров, используемых драйвером. Первая строка сообщает, что clang -ccl выполняет компиляцию от исходного кода на языке C до получения объектного кода. Последняя строка сообщает, что Clang все еще зависит от системного компоновщика, с помощью которого завершает компиляцию.

Внутренне, каждый вызов clang -ccl выполняет какую-то одну операцию. Полный список операций определен в исходном файле include/clang/Frontend/FrontendOptions.h. В табл. 4.1 перечислены некоторые операции с их описаниями, которые может выполнять clang -ccl:

Таблица 4.1. Некоторые операции, которые может выполнять clang -ccl

Операция	Описание
ASTView	Парсинг AST и вывод с помощью Graphviz
EmitBC	Вывод биткода LLVM в файл .bc
EmitObj	Вывод в объектный файл .o
FixIt	Парсинг и применение всех исправлений в исходном коде
PluginAction	Запуск операции расширения
RunAnalysis	Запуск одного или более видов анализа исходного кода

Параметр -ccl обеспечивает вызов функции ccl_main (подробности см. в файле tools/driver/ccl_main.cpp). Например, когда команда clang hello.c -o hello косвенно вызывает -ccl, эта функция инициализирует информацию о целевой архитектуре, настраивает инфраструктуру диагностики и выполняет операцию EmitObj. Эта

операция реализована в виде класса `CodeGenAction`, наследующего `FrontendAction`. Данный класс создает все необходимые экземпляры компонентов Clang и LLVM, и руководит ими в процессе создания объектного файла.

Наличие разных операций позволяет Clang использовать конвейер компиляции и для других нужд, например, для статического анализа. Однако, в зависимости от цели, определяемой в параметре `-target`, команда `clang` может загрузить другой объект `ToolChain`, определяющий, какие задачи должны выполняться компилятором `-cc1` для тех или иных операций, какие из них должны выполняться внешними инструментами и какие внешние инструменты должны использоваться. Например, для одной цели может использоваться GNU ассемблер и GNU компоновщик, для другой – интегрированный ассемблер LLVM и GNU компоновщик. Если вы сомневаетесь, какие внешние инструменты использует Clang для компиляции указанной цели, всегда можно воспользоваться ключом `-###` и получить список команд драйвера. Более подробно разные цели обсуждаются в главе 8, «Кросс-платформенная компиляция».

Библиотеки

С этого момента мы будем рассматривать Clang как множество библиотек, реализующих анализатор исходного кода, а не как драйвер и компилятор. В этом отношении Clang имеет модульную архитектуру и состоит из нескольких библиотек. Одним из важнейших интерфейсов Clang является библиотека `libclang` (http://clang.llvm.org/doxygen/group__CINDEX.html), которая содержит массу функций, доступных через C API. Она включает еще несколько библиотек Clang, которые могут использоваться отдельно в ваших проектах. Ниже приводится список библиотек, рассматриваемых в этой главе:

- `libclangLex`: используется для предварительной обработки и лексического анализа макросов, лексем и конструкций `pragma`;
- `libclangAST`: содержит функции для построения и управления абстрактных синтаксических деревьев;
- `libclangParse`: используется для парсинга результатов фазы лексического анализа;
- `libclangSema`: используется для семантического анализа, в ходе которого выполняется проверка AST;

- `libclangCodeGen`: генерирует код промежуточного представления LLVM IR с использованием информации о целевой архитектуре;
- `libclangAnalysis`: содержит ресурсы для статического анализа;
- `libclangRewrite`: обеспечивает поддержку и содержит инфраструктуру для реализации инструментов рефакторинга кода (подробнее об этом рассказывается в главе 10, «Инструменты Clang и фреймворк LibTooling»);
- `libclangBasic`: содержит множество утилит – абстракции управления памятью, поиск источников данных и диагностика.

Использование `libclang`

В этой главе мы будем описывать разные части анализатора исходного кода Clang и давать примеры использования C-интерфейса `libclang`. Даже при том, что это не C++ API, дающий непосредственный доступ к внутренним классам Clang, использование библиотеки `libclang` имеет свои преимущества, главным из которых является стабильность. Так как многие клиенты опираются на него, команда Clang поддерживает обратную совместимость интерфейса с предыдущими версиями. Однако ничто не мешает вам использовать обычные интерфейсы C++ LLVM, по аналогии с тем, как мы использовали интерфейсы C++ LLVM для чтения имен функций из биткода в главе 3, «Инструменты и организация».

В каталоге установки LLVM, в подкаталоге `include`, имеется подкаталог `clang-c`, где находятся заголовочные файлы библиотеки `libclang`. Для опробования примеров из этой главы, вам потребуется подключить файл `Index.h`, содержащий основные объявления точек входа в C-интерфейс Clang. Первоначально этот интерфейс создавался с целью упростить навигацию по исходным файлам и поддерживать механизмы поиска ошибок, дополнения кода и индексирования в интегрированной среде разработки, такой как Xcode, где главным заголовочным файлом считается файл `Index.h`. Ближе к концу главы мы также покажем, как использовать Clang через интерфейс C++.

В отличие от примеров в главе 3, «Инструменты и организация», где для создания списка библиотек LLVM на этапе сборки использовался инструмент `llvm-config`, у нас нет подобного инструмента для составления списка библиотек Clang. Чтобы определить правило компоновки с библиотекой `libclang`, можно взять за основу `Makefile` из главы 3, «Инструменты и организация», и изменить его, как показано

ниже. Как и прежде, не забудьте вручную вставить символы табуляции, чтобы обеспечить корректную обработку Makefile. Так как это типично для всех файлов сборки Makefile во всех примерах, отметьте, что мы используем флаг `llvm-config --libs` без каких-либо аргументов, который обеспечивает возврат полного списка библиотек LLVM.

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags)
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)

LANGLIBS = \
  -Wl,--start-group\
  -lclang\
  -lclangFrontend\
  -lclangDriver\
  -lclangSerialization\
  -lclangParse\
  -lclangSema\
  -lclangAnalysis\
  -lclangEdit\
  -lclangAST\
  -lclangLex\
  -lclangBasic\
  -Wl,--end-group

LLVMLIBS=$(shell $(LLVM_CONFIG) --libs)

PROJECT=myproject
PROJECT_OBJECTS=project.o

default: $(PROJECT)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET)$ (CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(PROJECT) : $(PROJECT_OBJECTS)
    @echo Linking $@
    $(QUIET)$ (CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ $(LANGLIBS)

$(LLVMLIBS)

clean:
    $(QUIET)rm -f $(PROJECT) $(PROJECT_OBJECTS)
```

Если вы используете динамические библиотеки и установили свою копию LLVM в нестандартное местоположение, вам будет недостаточно настроить переменную окружения `PATH`, так как динамическому компоновщику и загрузчику также потребуется знать, где находятся разделяемые библиотеки LLVM. Иначе, в момент запуска вашего проекта, они не смогут найти требуемые разделяемые библиотеки. В этом случае следует настроить путь поиска библиотек, как показано ниже:

```
$ export
LD_LIBRARY_PATH=$(LD_LIBRARY_PATH) :/путь_к_каталогу/установки/
llvm/lib
```

Замените `/путь_к_каталогу/установки/llvm` на полный путь к каталогу установки, как описывалось в главе 1, «Сборка и установка LLVM».

Диагностика в Clang

Диагностика является важнейшей частью взаимодействий компилятора с пользователями. Это – сообщения, которые возвращает компилятор, содержащие описания ошибок, предупреждения или подсказки. Clang имеет богатый механизм диагностики с весьма информативными сообщениями об ошибках. Внутренне Clang делит диагностические сообщения на типы: каждый этап анализа исходного кода имеет свое множество диагностических сообщений определенного типа. Например, диагностические сообщения для этапа парсинга определены в файле `include/clang/Basic/DiagnosticParseKinds.td`. Анализатор Clang также классифицирует диагностические сообщения по серьезности проблем: `NOTE`, `WARNING`, `EXTENSION`, `EXTWARN` и `ERROR`. Эта классификация определена как перечисление `Diagnostic::Level`.

Вы можете создавать собственные диагностические сообщения, добавляя новые определения TableGen в виде файлов `include/clang/Basic/Diagnostic*Kinds.td`, а также код, выполняющий проверки требуемых условий и выводящий соответствующие диагностические сообщения. Все файлы с расширением `.td`, что находятся в дереве исходных текстов LLVM, написаны на языке TableGen.

TableGen – это один из инструментов в составе LLVM, используемый системой сборки LLVM, чтобы сгенерировать программный код на C++ для тех частей компилятора, которые могут быть синтезированы механическим способом. Эта идея родилась в генераторах выполняемого кода LLVM, где имеется масса кода, который

можно сгенерировать, опираясь на описание целевой архитектуры, и ныне широко используется в проекте LLVM. Язык TableGen создан для представления информации в виде записей. Например, файл `DiagnosticParseKinds.td` содержит следующие записи:

```
def err_invalid_sign_spec : Error<"'%0' cannot be signed or unsigned">;
def err_invalid_short_spec : Error<"'short %0' is invalid">;
```

Здесь `def` – это ключевое слово языка TableGen, определяющее новую запись. Число и типы полей в записях зависят от того, какой генератор кода TableGen будет использоваться, и для каждого типа генерируемых файлов существует свой генератор. В результате обработки определений на языке TableGen всегда создается файл `.inc`, который подключается другими исходными файлами LLVM. В данном случае TableGen сгенерирует `DiagnosticsParseKinds.inc` с макроопределениями для каждого диагностического сообщения.

`err_invalid_sign_spec` и `err_invalid_short_spec` – это идентификаторы записей, а `Error` – класс TableGen. Имейте в виду, что смысл терминов здесь несколько отличается от похожих терминов в языке C++. Каждый класс в языке TableGen, не путайте с классами в C++, – это шаблон записи, определяющий поля, которые наследуются другими записями. Так же как в C++, классы в языке TableGen могут образовывать иерархии.

Для определения параметров в определениях используется синтаксис шаблонов на основе класса `Error`, который принимает единственную строку. Все определения, основанные на этом классе, будут преобразованы в диагностические сообщения типа `ERROR` с текстом, переданным в параметре класса, например, `"'short %0' is invalid"`. Несмотря на свою простоту, синтаксис языка TableGen может вводить читателей в заблуждение из-за большого объема информации, содержащейся в записях. В случае каких-либо сомнений, обращайтесь к документации <http://llvm.org/docs/TableGen/LangRef.html>.

Чтение диагностических сообщений

Далее мы рассмотрим пример на языке C++, использующий библиотеку `libclang` для чтения и вывода всех диагностических сообщений, произведенных анализатором Clang при чтении заданного исходного файла.

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
```

```

#include <iostream>

using namespace llvm;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"), cl::Required);

int main(int argc, char** argv)
{
    cl::ParseCommandLineOptions(argc, argv, "Diagnostics Example");
    CXIndex index = clang_createIndex(0, 0);
    const char *args[] = {
        "-I/usr/include",
        "-I."
    };
    CXTranslationUnit translationUnit = clang_parseTranslationUnit
        (index, FileName.c_str(), args, 2, NULL, 0,
         CXTranslationUnit_None);
    unsigned diagnosticCount = clang_getNumDiagnostics(translationUnit);
    for (unsigned i = 0; i < diagnosticCount; ++i) {
        CXDiagnostic diagnostic = clang_getDiagnostic(translationUnit, i);
        CXString category = clang_getDiagnosticCategoryText(diagnostic);
        CXString message = clang_getDiagnosticSpelling(diagnostic);
        unsigned severity = clang_getDiagnosticSeverity(diagnostic);
        CXSourceLocation loc = clang_getDiagnosticLocation(diagnostic);
        CXString fName;
        unsigned line = 0, col = 0;
        clang_getPresumedLocation(loc, &fName, &line, &col);
        std::cout << "Severity: " << severity << " File: "
                  << clang_getCString(fName) << " Line: "
                  << line << " Col: " << col << " Category: \""
                  << clang_getCString(category) << "\" Message: "
                  << clang_getCString(message) << std::endl;
        clang_disposeString(fName);
        clang_disposeString(message);
        clang_disposeString(category);
        clang_disposeDiagnostic(diagnostic);
    }
    clang_disposeTranslationUnit(translationUnit);
    clang_disposeIndex(index);
    return 0;
}

```

Подключение заголовочного C-файла библиотеки `libclang` в файле с исходным кодом на C++ выполнено в окружении `extern "C"`, чтобы компилятор C++ скомпилировал его как исходный код на C.

Как и в предыдущей главе, мы вновь использовали пространство имен `cl`, чтобы получить возможность парсинга аргументов командной строки. Затем мы использовали несколько функций из интер-

фейса `libclang` (http://clang.llvm.org/doxygen/group_CINDEX.html). Сначала вызовом функции `clang_createIndex()` мы создали индекс, структуру контекста, используемую библиотекой `libclang`. Эта функция принимает два булевых параметра в виде целых чисел: в первом следует передать `true`, если требуется исключить объявления из **предварительно скомпилированных заголовочных файлов (Precompiled Headers, PCH)**, а во втором следует передать `true`, если требуется вывести диагностические сообщения. В обоих параметрах мы передали `false` (ноль), потому что выводить диагностические сообщения мы будем сами.

Далее мы обратились к анализатору Clang, чтобы выполнить парсинг единицы трансляции, вызвав функцию `clang_parseTranslationUnit()` (см. http://clang.llvm.org/doxygen/group_CINDEX_TRANSLATION_UNIT.html). Она принимает имя исходного файла для парсинга, которое в данном примере хранится в глобальной переменной `FileName`. Данная переменная получает значение из строкового параметра командной строки, переданного программе в момент ее запуска. Функции также нужно передать два аргумента, определяющих, где искать заголовочные файлы – вы можете изменить их значения, в соответствии с вашей системой.

Совет. *Самой сложной частью реализации собственного инструмента Clang является необходимость угадывать параметры для драйвера, которые обеспечивали бы корректную обработку исходных файлов в вашей системе. Об этом не пришлось бы беспокоиться, если бы создавалось, к примеру, расширение для Clang. Для решения этой проблемы можно использовать базу данных команд компиляции, обсуждаемую в главе 10, «Инструменты Clang и фреймворк LibTooling», где хранятся точные наборы параметров для обработки каждого исходного файла, который требуется проанализировать. В данном случае базу данных можно было бы сгенерировать с помощью CMake. Однако мы предпочли передать аргументы явно.*

После парсинга и сохранения всей информации в структуре `CXTranslationUnit`, выполняется обход всех диагностических сообщений, сгенерированных анализатором Clang, и вывод их на экран. Для этого сначала вызывается функция `clang_getNumDiagnostics()`, извлекающая число диагностических сообщений, и определяются границы цикла (см. http://clang.llvm.org/doxygen/group_CINDEX_DIAG.html). Затем в каждой итерации вызывается функция `clang_getDiagnostic()`, извлекающая текущее сообще-

ние, `clang_getDiagnosticCategoryText()`, извлекающая строку с описанием типа сообщения, `clang_getDiagnosticSpelling()`, извлекающая текст сообщения для вывода на экран и `clang_getDiagnosticLocation()`, извлекающая точное местоположение в исходном коде. Мы также использовали функцию `clang_getDiagnosticSeverity()`, чтобы получить элемент перечисления, представляющий важность данного сообщения (`NOTE`, `WARNING`, `EXTENSION`, `EXTWARN` или `ERROR`), но для простоты преобразуем его в беззнаковое целое и выводим как простое число.

Так как мы используем C-интерфейс, где отсутствует класс `string`, функции, возвращающие строки, часто возвращают их в виде специального объекта `CXString`, поэтому мы вынуждены использовать функцию `clang_getCString()` для доступа к внутреннему буферу со строкой, и `clang_disposeString()`, чтобы потом удалить объекты.

Не забывайте, что анализируемый исходный файл может подключать другие файлы, поэтому помимо номера строки и позиции в строке механизм диагностики должен также выводить имя файла. Тройка атрибутов – имя файла, номер строки в файле и номер позиции в строке – позволят найти код, к которому относится то или иное диагностическое сообщение. Эту тройку атрибутов представляет специальный объект `CXSourceLocation`. Чтобы извлечь из этого объекта имя файла, номер строки в файле и номер позиции в строке, необходимо вызвать функцию `clang_getPresumedLocation()`, принимающую объект `CXString` и переменные типа `int` по ссылке, которые она заполнит информацией из объекта `CXSourceLocation`.

По окончании мы удаляем объекты вызовом `clang_disposeDiagnostic()`, `clang_disposeTranslationUnit()` и `clang_disposeIndex()`.

Теперь проверим работу нашего инструмента, передав ему следующий файл `hello.c`:

```
int main() {
    printf("hello, world!\n")
}
```

В этом файле допущены две ошибки: отсутствует подключение соответствующего заголовочного файла и точка с запятой после инструкции `printf`. Соберем наш проект и запустим его, чтобы увидеть, какие диагностические сообщения он нам вернет:

```
$ make
$ ./myproject hello.c
Severity: 2 File: hello.c Line: 2 Col: 9 Category: "Semantic Issue"
```

```
Message: implicitly declaring library function 'print' with type
'int (const char *, ...)'
```

```
Severity: 3 File: hello.c Line: 2 Col: 24 Category: "Parse Issue"
Message: expected ';' after expression
```

Как видите, на разных этапах работы анализатора (семантический анализ и парсинг) было сгенерировано два диагностических сообщения. Каждый из этапов мы исследуем в следующих разделах.

Этапы работы анализатора Clang

Преобразование исходного текста программы в биткод LLVM IR выполняется в несколько этапов. Все они изображены на рис. 4.1, и все они будут рассматриваться в этом разделе.

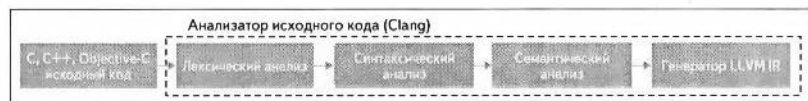


Рис. 4.1. Этапы преобразования исходного текста программы в биткод LLVM IR

Лексический анализ

На самом первом этапе анализатор исходного кода разбивает языковые конструкции в текстовом файле на множество слов и лексем, удаляя такие элементы программ, как комментарии, пробелы и табуляции. Каждое слово или лексема должно принадлежать подмножеству языка, а каждое зарезервированное слово преобразуется во внутреннее представление компилятора. Зарезервированные слова определены в файле `include/clang/Basic/TokenKinds.def`. Например, взгляните, как определены зарезервированное слово `while` и символ `<`, две известных лексемы языка C/C++ (мы выделили их жирным в следующей ниже выдержке из файла `TokenKinds.def`):

```
ТОК(identifier)           // abcde123
// Строковые литералы C++11.
ТОК(utf32_string_literal) // U"foo"
...
PUNCTUATOR(r_paren,      ")")
PUNCTUATOR(l_brace,     "{")
PUNCTUATOR(r_brace,     "}")
```

```

PUNCTUATOR(starequal,      "*=")
PUNCTUATOR(plus,          "+")
PUNCTUATOR(plusplus,      "++")
PUNCTUATOR(arrow,         "->")
PUNCTUATOR(minusminus,    "--")
PUNCTUATOR(less,         "<")
...
KEYWORD(float,             , KEYALL)
KEYWORD(goto,              , KEYALL)
KEYWORD(inline,           , KEYC99|KEYCXX|KEYGNU)
KEYWORD(int,               , KEYALL)
KEYWORD(return,           , KEYALL)
KEYWORD(short,            , KEYALL)
KEYWORD(while,           , KEYALL)

```

Определения из этого файла помещаются в пространство имен tok. То есть, если компилятору потребуется проверить присутствие зарезервированных слов после лексического анализа, он сможет сделать это с помощью данного пространства имен. Например, конструкции {, <, goto и while доступны как элементы перечисления tok::l_brace, tok::less, tok::kw_goto и tok::kw_while.

Взгляните на следующий код на языке C в файле min.c:

```

int min(int a, int b) {
    if (a < b)
        return a;
    return b;
}

```

Каждая лексема содержит экземпляр класса SourceLocation, который хранит местоположение лексемы в исходном тексте программы. Выше мы уже использовали его аналог CXSourceLocation на языке C, — оба они хранят одни и те же данные. Мы можем вывести все лексемы и их местоположения (SourceLocation) после лексического анализа следующей командой:

```
$ clang -cc1 -dump-tokens min.c
```

Например, для инструкции if в примере выше мы получим:

```

if 'if' [StartOfLine] [LeadingSpace] Loc=<min.c:2:3>
l_paren '(' [LeadingSpace] Loc=<min.c:2:6>
identifier 'a' Loc=<min.c:2:7>
less '<' [LeadingSpace] Loc=<min.c:2:9>
identifier 'b' [LeadingSpace] Loc=<min.c:2:11>
r_paren ')' Loc=<min.c:2:12>
return 'return' [StartOfLine] [LeadingSpace] Loc=<min.c:3:5>
identifier 'a' [LeadingSpace] Loc=<min.c:3:12>
semi ';' Loc=<min.c:3:13>

```

Обратите внимание, что каждая языковая конструкция снабжается префиксом, определяющим ее тип: `r_paren` для `)`, `less` для `<`, `identifier` для строк, не соответствующих зарезервированным словам, и так далее.

Проверка наличия лексических ошибок

Рассмотрим исходный код в файле `lex-err.c`:

```
int a = 08000;
```

Ошибка здесь заключается в неправильном оформлении восьмеричной константы: восьмеричные константы не должны включать цифры выше 7. Это определение вызывает следующую лексическую ошибку:

```
$ clang -c lex.c
lex.c:1:10: error: invalid digit '8' in octal constant
int a = 08000;
          ^
```

```
1 error generated.
```

Теперь проверим этот файл с помощью программы, созданной в предыдущем разделе:

```
$ ./myproject lex.c
```

```
Severity: 3 File: lex.c Line: 1 Col: 10 Category: "Lexical or Preprocessor Issue" Message: invalid digit '8' in octal constant
```

Как видите, наша программа тоже выявила лексическую ошибку.

Использование libclang для лексического анализа

Ниже приводится пример использования библиотеки `libclang` для выделения лексем с помощью лексического анализатора LLVM, в первых 60 символах в файле с исходным кодом:

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
#include <iostream>

using namespace llvm;
static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"),
         cl::Required);

int main(int argc, char** argv)
```

```

{
    cl::ParseCommandLineOptions(argc, argv, "My tokenizer\n");
    CXIndex index = clang_createIndex(0,0);
    const char *args[] = {
        "-I/usr/include",
        "-I."
    };
    CXTranslationUnit translationUnit = clang_parseTranslationUnit(
        index, FileName.c_str(), args,
        2, NULL, 0, CXTranslationUnit_None);
    CXFile file = clang_getFile(translationUnit, FileName.c_str());
    CXSourceLocation loc_start = clang_getLocationForOffset(translationUnit,
        file, 0);
    CXSourceLocation loc_end = clang_getLocationForOffset(translationUnit,
        file, 60);
    CXSourceRange range = clang_getRange(loc_start, loc_end);
    unsigned numTokens = 0;
    CXToken *tokens = NULL;
    clang_tokenize(translationUnit, range, &tokens, &numTokens);
    for (unsigned i = 0; i < numTokens; ++i) {
        enum CXTokenKind kind = clang_getTokenKind(tokens[i]);
        CXString name = clang_getTokenSpelling(translationUnit, tokens[i]);
        switch (kind) {
            case CXToken_Punctuation:
                std::cout << "PUNCTUATION(" << clang_getCString(name) << ") ";
                break;
            case CXToken_Keyword:
                std::cout << "KEYWORD(" << clang_getCString(name) << ") ";
                break;
            case CXToken_Identifier:
                std::cout << "IDENTIFIER(" << clang_getCString(name) << ") ";
                break;
            case CXToken_Literal:
                std::cout << "COMMENT(" << clang_getCString(name) << ") ";
                break;
            default:
                std::cout << "UNKNOWN(" << clang_getCString(name) << ") ";
                break;
        }
        clang_disposeString(name);
    }
    std::cout << std::endl;
    clang_disposeTokens(translationUnit, tokens, numTokens);
    clang_disposeTranslationUnit(translationUnit);
    return 0;
}

```

Анализ начинается по тому же шаблону, что и выше – выполняет-ся инициализация параметров командной строки и вызывается пара функций `clang_createIndex()/clang_parseTranslationUnit()`.

Дальше начинаются отличия. Вместо обращения к механизму диагностики мы готовим аргументы для функции `clang_tokenize()`, которая выполнит лексический анализ и вернет поток лексем. Для этого мы создаем объект `CXSourceRange`, указав диапазон исходного кода (начало и конец) для анализа. Этот объект можно сконструировать из двух объектов `CXSourceLocation`, из которых один обозначает начало, а другой – конец. Мы создаем их с помощью функции `clang_getLocationForOffset()`, возвращающей объект `CXSourceLocation` для заданного смещения в объекте `CXFile`, полученном вызовом `clang_getFile()`.

Чтобы создать `CXSourceRange` из двух объектов `CXSourceLocation`, вызывается функция `clang_getRange()`. После этого все готово для вызова `clang_tokenize()` с двумя параметрами, которые передаются по ссылке: указатель на `CXToken`, для сохранения потока лексем, и переменная типа `unsigned`, в которой возвращается число лексем в потоке. На основании этого числа конструируется цикл, выполняющий итерации по всем лексемам.

Для каждой лексемы мы получаем ее тип, вызовом `clang_getTokenKind()`, и соответствующий фрагмент кода, вызовом `clang_getTokenSpelling()`. Затем с помощью конструкции выбора `switch` выводятся разные текстовые сообщения, в зависимости от типа лексемы, и фрагмент кода, соответствующий данной лексеме. Результаты анализа вы можете видеть ниже.

На вход программы мы передали следующий исходный код:

```
#include <stdio.h>
int main() {
    printf("hello, world!");
}
```

После запуска наша программа вывела следующее:

```
PUNCTUATION(#) IDENTIFIER(include) PUNCTUATION(<) IDENTIFIER(stdio)
PUNCTUATION(.) IDENTIFIER(h) PUNCTUATION(>) KEYWORD(int)
IDENTIFIER(main) PUNCTUATION(()) PUNCTUATION(()) PUNCTUATION(())
IDENTIFIER(printf) PUNCTUATION(()) COMMENT("hello, world!")
PUNCTUATION(()) PUNCTUATION(;) PUNCTUATION(())
```

Препроцессор

Препроцессор C/C++ выполняется перед семантическим анализом и отвечает за развертывание макросов, подключение файлов и пропускает фрагменты кода посредством директив препроцессора, которые начинаются с символа `#`. Препроцессор работает в тесном

сотрудничестве с лексическим анализатором – они непрерывно взаимодействуют друг с другом. Так как он выполняется на раннем этапе, в анализаторе исходного кода и перед тем, как в ходе семантического анализа будет предпринята попытка оценить смысл программного кода, препроцессор может самым причудливым образом влиять на код, например, изменять объявления функций при разворачивании макросов. Имейте в виду, что эта особенность позволяет радикально менять синтаксис языка. При желании вы сможете даже написать такой код, как показано на рис. 4.2.

```

#include "SDL.h"

#define $ for(0*9
#define CX M+=IT*3+2*!(IT+6)
#define x ,A=4*-1,U=t,W=n=i+J+r(Q/p:U(A+J),U(A),U(A+1)[1]+U(A+2)*g+):K(I),U=V=K(a),o
    N=n,W=V:V,
#define C B+~L
#define Z short
#define y a(Z)Y{++0}
#define B ),a--|{|
#define _ ),e--|{|

#define V(I,D,E)(0=a(I)n[r])&&!(A=(0)(V=[E+L]<<16)+*1)/0,A=-(I)A?1[E+L]=V-0*(+E=A
    ):H(0)
#define i(B,M)B(o){return M;}
#define R(I,O,M, )({S=L?a(I Z):0:0,N=i?a(I Z)0 M(f=a(I Z)):0 M(f=a(I n))})
#define T()R(r(lu(10),L=4,--),,_)
#define u(a,r,T)16*i[a]+(I Z){T i[r]}
#define a( )+(**)$
#define L()M(W,_,U)

#define M(S,F,T)R(r[S],F,r[T])
#define A( )i{l=4}+=2,R(,_,r[u(10,4,-2+)]})
#define c(R,T){l=um9,L+T=(N=a(R)h[r]+(R)*T)>>16,*i=N,G(FIN-(R)N)}
#define hi( )!L(L?a(Z):_)>>C-1}
#define I unsigned
#define n char
#define ei( )v(F[40[L[_#]=40[E]+],E]&N=#S[_ N<_int)5])

I n t,e,l 80186),=E,m,u,L,a,T,o,r{1<<21},X,*Y,b,0=0,R=0;I Z=i,M,p,q=3;I+localtime(),
T,S,kb=8,h,W,U,c,g,d,V,A;N,O,P=983040,[5];SDL_Surface*k=0;i(K,P+(L?2*o:2*o+o/4$
7))i(B,r(a(I)E[259+4*o]+0))i(w,i[o]+~(-2+47[E])~L)i(v,(z((f=*N)&16),G(N-56$1
&40[E]^>>C-1)))}({V=61442;S;0--;V+=40[E+0]<<0(25);}i(H,(46[u=76,J(i),T(V),T
(9[1]),T(M),M(P+18,=,4+o+2),R(M,=,r[4*o]),E)=0)}s(o){S;0--;40[E-0]=3&&1<<0(25)&
o;}i(BP,(w+i=262+o+z(F(=E15)>0|42[E]),=E&15)}i(SP,(w(7),R&E--i[i]&6o7R+.,Q&E
Q+*,M--:0))DX(}{S,0=27840;0--;10[(I)k->pixels]=!{1<<7-0*8&R[0/2880=90+0%720/
8+(88+952[l]/128+4+0/720%4<<13)});SDL_Flip(k);}main(BX,nE)n=nE;{9(i=E+r+P)=P>>4
;S;q;l[--q]=++nE?open(*nE,32898):8;read(2[a(I)=i+*]?lseek(+j,0,2)>>9:0,)},E+(M
=256),P);S;Y=r+16*9[1]+M,Y-r;Q|R}|kt&46[E]&&KB)--64[T=1[0=32[L=(X=Y&7)&1,o=X/2$
1,l]=0,t=(c=y)&7,a=c/8&7,Y>>6,g=~T?y:(n)y,d=BXy,l,!T+~66&T-27T-17dmg:0:(dny
),Q&50--,R&6R--x(0=+Y,0=U=0(51),e=D(8),m=D(14)_ 0=*Y/2&7,M+(n)c=(L^D(m)[E]D
(22)[E]D(23)[E]^D(24)[E]))_ L=Y&8,R(K(X)[r],,c)_ L=+3,o=0,a=X x a=m _ T(X[i

```

Рис. 4.2. Макросы позволяют радикально менять синтаксис языка

Этот код написал Адриан Кейбл (Adrian Cable), один из победителей 22-го международного конкурса запутывания кода на C (International Obfuscated C Code Contest, IOCCC), который к нашей радости позволил воспроизвести конкурсную разработку на услови-

ях лицензии Creative Commons Attribution-ShareAlike 3.0 License. Это – эмулятор 8086. Если вы хотите узнать, как привести этот код к более привычному виду, прочитайте раздел «ClangFormat» в главе 10, «Инструменты Clang и фреймворк LibTooling». Развернуть макросы также можно, вызвав драйвер компилятора с флагом `-E`, который прерывает компиляцию сразу после завершения препроцессора.

Тот факт, что макросы позволяют превратить исходный код в невразумительный текст, может служить предупреждением, что к ним следует относиться с осторожностью. Поток лексем обрабатывается препроцессором на этапе лексического анализа, чтобы развернуть директивы, такие как макросы и прагмы. Определения макросов сохраняются препроцессором в таблице символов и, всякий раз, когда в исходном коде встречается макрос, лексемы из таблицы символов замещают текущие.

Если установить дополнительные инструменты Clang (глава 2, «Внешние проекты»), в вашем распоряжении появится утилита `pp-trace`. Этот инструмент позволяет исследовать активность препроцессора.

Рассмотрим следующий пример, файл `pp.c`:

```
#define EXIT_SUCCESS 0
int main() {
    return EXIT_SUCCESS;
}
```

Если запустить драйвер компилятора с параметром `-E`, он выведет следующее:

```
$ clang -E pp.c -o pp2.c && cat pp2.c
...
int main() {
    return 0;
}
```

Если запустить инструмент `pp-trace`:

```
$ pp-trace pp.c
...
- Callback: MacroDefined
  MacroNameTok: EXIT_SUCCESS
  MacroDirective: MD_Define
- Callback: MacroExpands
  MacroNameTok: EXIT_SUCCESS
  MacroDirective: MD_Define
  Range: ["/examples/pp.c:3:10", "/examples/pp.c:3:10"]
  Args: (null)
- Callback: EndOfFile
```


Мы опустили длинный список встроенных макросов, которые `pp-trace` выводит перед началом обработки файла. В действительности этот список может очень пригодиться, если вы пожелаете узнать, какие макросы определяются драйвером компилятора по умолчанию. Реализация инструмента `pp-trace` использует прием переопределения функций обратного вызова препроцессора, это означает, вы сможете реализовать в своем инструменте функции, которые будут вызываться при каждом проявлении препроцессора. В нашем примере, препроцессор проявил себя дважды: первый раз он прочитал определение макроса `EXIT_SUCCESS` и затем развернул этот макрос в строке 3. Инструмент `pp-trace` также вывел параметры, которые он принимает, если реализует функцию обратного вызова `MacroDefined`. Инструмент имеет очень небольшой размер, и если вы пожелаете реализовать функции обратного вызова для препроцессора, чтение исходных текстов этого инструмента послужат вам отличной отправной точкой.

Синтаксический анализ

После разделения исходного кода на лексемы в ходе лексического анализа, выполняется синтаксический анализ, объединяющий лексемы в выражения, инструкции и тела функций. В ходе этого анализа проверяется, имеют ли смысл получившиеся группы лексем, с учетом их физического расположения, но смысл самого кода пока не анализируется, так же как синтаксический анализ предложений на языке межчеловеческого общения не заботится о смысловой нагрузке предложений, а только лишь проверяет – насколько правильно они построены. Этот вид анализа также называют парсингом (*parsing*) – он получает на входе поток лексем и выводит абстрактное синтаксическое дерево (*Abstract Syntax Tree, AST*).

Узлы дерева AST

Узлы абстрактного синтаксического дерева представляют объявления, инструкции и типы. Соответственно существует три базовых класса для представления узлов дерева *AST*: `Decl`, `Stmt` и `Type`. Каждая конструкция языка *C* или *C++* представляется в *Clang* классом *C++*, который должен наследовать один из трех базовых классов. На рис. 4.3 показана часть иерархии классов. Например, класс `IfStmt` (представляющий тело инструкции `if`) непосредственно наследует класс `Stmt`. Классы `FunctionDecl` и `VarDecl`, используемые для хранения объявлений функций и переменных, с другой стороны, наследуют несколько классов и являются непрямymi потомками класса `Decl`.

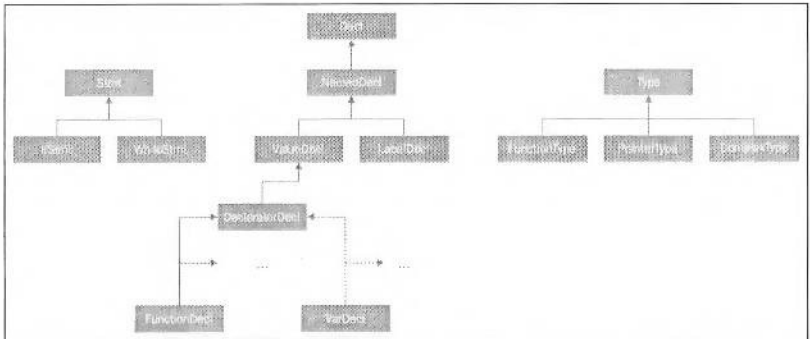


Рис. 4.3. Часть иерархии классов, представляющих конструкции языка C или C++

Полную иерархию классов можно найти в документации doxygen, доступной в Интернете. Например, иерархию для класса Stmt можно увидеть на странице http://clang.llvm.org/doxygen/classclang_1_1Stmt.html – вы можете щелкать на именах классов, чтобы увидеть их непосредственных наследников.

Узлом AST самого верхнего уровня является экземпляр TranslationUnitDecl. Это корень всего дерева AST и представляет единицу трансляции. Воспользуемся еще раз содержимым файла min.c и посмотрим, как выглядит дерево AST для этого файла, которое можно получить с помощью ключа `-ast-dump`:

```

$ clang -fsyntax-only -Xclang -ast-dump min.c
TranslationUnitDecl ...
|-TypedefDecl ... __int128_t '__int128'
|-TypedefDecl ... __uint128_t 'unsigned __int128'
|-TypedefDecl ... __builtin_va_list '__va_list_tag [1]'
'-FunctionDecl ... <min.c:1:1, line:5:1> min 'int (int, int)'
  |-ParmVarDecl ... <line:1:7, col:11> a 'int'
  |-ParmVarDecl ... <col:14, col:18> b 'int'
  '-CompoundStmt ... <col:21, line:5:1>
...

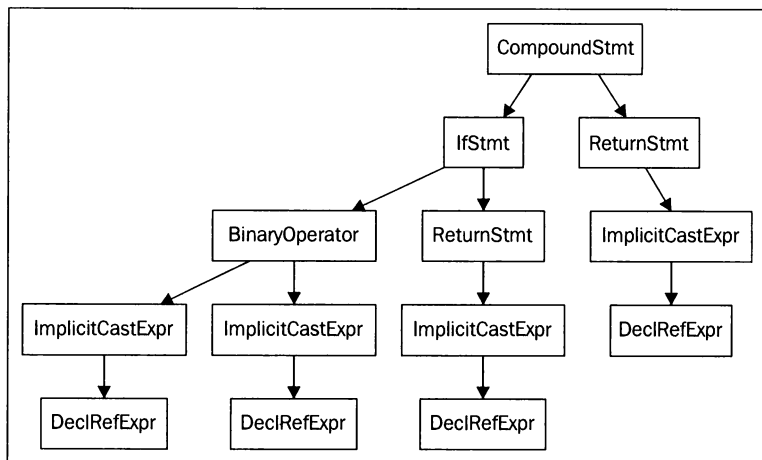
```

Обратите внимание на корневой узел TranslationUnitDecl, представляющий единицу трансляции, и объявление функции min, представленное экземпляром FunctionDecl. Объявление CompoundStmt содержит другие инструкции и выражения. В графическом виде дерево AST изображено на рис. 4.4, полученном командой:

```

$ clang -fsyntax-only -Xclang -ast-view min.c

```



Function body AST from min.c

Рис. 4.4. Дерево AST для файла min.c

. Узел `CompoundStmt` содержит инструкции `if` и `return` (`IfStmt` и `ReturnStmt`, соответственно). При каждом использовании переменных `a` и `b` генерируется выражение `ImplicitCastExpr` для типа `int`, как того требуют стандарты языка C.

Все дерево AST для единицы трансляции хранится в экземпляре класса `ASTContext`. Начав навигацию с экземпляра `TranslationUnitDecl`, доступного через вызов `ASTContext::getTranslationUnitDecl()`, можно добраться до любого узла AST.

Исследование работы парсера с помощью отладчика

Множество лексем, созданных на этапе лексического анализа, обрабатываются затем на этапе парсинга и на их основе создаются узлы дерева AST. Например, всякий раз, когда обнаруживается лексема `tok::kw_if`, вызывается функция `ParseIfStatement`, которая извлекает из потока все лексемы, образующие тело инструкции `if`, генерирует все необходимые дочерние узлы AST и корневой узел `IfStmt` для них. Взгляните на следующий фрагмент из файла `lib/Parse/ParseStmt.cpp` (строка 212):

```

...
case tok::kw_if: // C99 6.8.4.1: if-statement
    return ParseIfStatement(TrailingElseLoc);

```



```

case tok::kw_switch: // C99 6.8.4.2: switch-statement
    return ParseSwitchStatement(TrailingElseLoc);
...

```

Чтобы лучше понять, как Clang добирается до метода `ParseIfStatement` при анализе файла `min.c`, выведем дампы трассировки вызовов в отладчике:

```

$ gdb clang

$ b ParseStmt.cpp:213

$ r -ccl -fsyntax-only min.c
...
213 return ParseIfStatement(TrailingElseLoc);

(gdb) backtrace
#0 clang::Parser::ParseStatementOrDeclarationAfterAttributes
#1 clang::Parser::ParseStatementOrDeclaration
#2 clang::Parser::ParseCompoundStatementBody
#3 clang::Parser::ParseFunctionStatementBody
#4 clang::Parser::ParseFunctionDefinition
#5 clang::Parser::ParseDeclGroup
#6 clang::Parser::ParseDeclOrFunctionDefInternal
#7 clang::Parser::ParseDeclarationOrFunctionDefinition
#8 clang::Parser::ParseExternalDeclaration
#9 clang::Parser::ParseTopLevelDecl
#10 clang::ParseAST
#11 clang::ASTFrontendAction::ExecuteAction
#12 clang::FrontendAction::Execute
#13 clang::CompilerInstance::ExecuteAction
#14 clang::ExecuteCompilerInvocation
#15 ccl_main
#16 main

```

Парсинг единицы трансляции начинается с вызова функции `ParseAST()`, которая читает объявления верхнего уровня вызовом `Parser::ParseTopLevelDecl()`. Затем она извлекает лексемы из потока и генерирует соответствующие узлы AST, присоединяя каждый новый узел AST к его родительскому узлу. Управление возвращается в функцию `ParseAST()`, только когда из потока будут извлечены все лексемы. После этого пользователь парсера может обращаться к узлам AST через корневой узел типа `TranslationUnitDecl`.

Проверка наличия синтаксических ошибок

Рассмотрим следующую инструкцию `for` в файле `parse.c`:

```

void func() {
    int n;

```

```
    for (n = 0; n < 10; n++);
}
```

Этот код содержит ошибку – после выражения `n = 0` отсутствует точка с запятой. Ниже приводится диагностическое сообщение, которое Clang выводит в процессе компиляции:

```
$ clang -c parse.c
```

```
parse.c:3:14: error: expected ';' in 'for' statement specifier
    for (n = 0; n < 10; n++);
                   ^
```

```
1 error generated.
```

Теперь запустим наш проект, осуществляющий диагностику:

```
$ ./myproject parse.c
```

```
Severity: 3 File: parse.c Line: 3 Col: 14 Category: "Parse Issue"
Message: expected ';' in 'for' statement specifier
```

Так как все лексемы в данном примере верны, лексический анализ завершается благополучно, без диагностических сообщений. Однако, когда при создании дерева AST выполняется группировка лексем, чтобы увидеть, имеют ли они смысл, парсер обнаруживает, что в структуре `for` отсутствует точка с запятой. В данном случае было создано диагностическое сообщение из категории *Parse Issue* (ошибка парсинга).

Обход дерева AST программным способом

Интерфейс `libclang` позволяет осуществлять обход дерева Clang AST с помощью объекта-курсора, указывающего на узел текущего дерева AST. Получить курсор верхнего уровня можно вызовом функции `clang_getTranslationUnitCursor()`. В этом разделе мы напишем инструмент, который выводит имена всех функций или методов, имеющихся в файле с исходным кодом:

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
#include <iostream>

using namespace llvm;
static cl::opt<std::string>

FileName(cl::Positional, cl::desc("Input file"), cl::Required);

enum CXChildVisitResult visitNode (CXCursor cursor, CXCursor parent,
```

```

CXClientData client_data) {
    if (clang_getCursorKind(cursor) == CXCursor_CXXMethod ||
        clang_getCursorKind(cursor) == CXCursor_FunctionDecl) {
        CXString name = clang_getCursorSpelling(cursor);
        CXSourceLocation loc = clang_getCursorLocation(cursor);
        CXString fName;
        unsigned line = 0, col = 0;
        clang_getPresumedLocation(loc, &fName, &line, &col);
        std::cout << clang_getCString(fName) << ":"
            << line << ":" << col << " declares "
            << clang_getCString(name) << std::endl;
        return CXChildVisit_Continue;
    }
    return CXChildVisit_Recurse;
}

int main(int argc, char** argv)
{
    cl::ParseCommandLineOptions(argc, argv, "AST Traversal Example");
    CXIndex index = clang_createIndex(0, 0);
    const char *args[] = {
        "-I/usr/include",
        "-I."
    };
    CXTranslationUnit translationUnit =
        clang_parseTranslationUnit(index, FileName.c_str(), args, 2, NULL, 0,
                                   CXTranslationUnit_None);
    CXCursor cur = clang_getTranslationUnitCursor(translationUnit);
    clang_visitChildren(cur, visitNode, NULL);
    clang_disposeTranslationUnit(translationUnit);
    clang_disposeIndex(index);
    return 0;
}

```

Наиболее важной в этом примере является функция `clang_visitChildren()`, которая рекурсивно посещает все дочерние узлы курсора, переданного ей в виде параметра, и вызывает указанную функцию обратного вызова. Пример начинается с определения этой функции обратного вызова, `visitNode()`. Данная функция должна возвращать элемент перечисления `CXChildVisitResult`, что дает нам три возможности:

- вернуть `CXChildVisit_Recurse`, когда требуется, чтобы `clang_visitChildren()` продолжила обход дерева AST и посетила узлы, являющиеся дочерними по отношению к текущему;
- вернуть `CXChildVisit_Continue`, когда требуется продолжить обход, но пропустить узлы, являющиеся дочерними по отношению к текущему;

- вернуть `CXChildVisit_Break`, когда требуется, чтобы `clang_visitChildren()` прекратила обход узлов.

Функция обратного вызова принимает три параметра: курсор, представляющий текущий узел AST; еще один курсор, представляющий родителя текущего узла; и объект `CXClientData`, который определен через `typedef` как указатель типа `void`. Этот указатель позволяет передавать любые структуры данных, необходимые для работы функции обратного вызова. Это может пригодиться, например, для проведения дополнительного анализа.

Совет. Данную структуру можно использовать для организации дополнительного анализа, но, если окажется, что анализ слишком сложен и требует передачи такой структуры, как **граф управляющей логики (Control Flow Graph, CFG)**, не используйте курсоры или библиотеку `libclang` – более адекватным подходом в такой ситуации является реализация анализа в виде расширения для `Clang`, которое непосредственно использует `Clang C++ API` для создания CFG на основе AST (см. <http://clang.llvm.org/docs/ClangPlugins.html> и метод `CFG::buildCFG`). Конструировать дополнительные виды анализа непосредственно на основе дерева AST обычно сложнее, чем с применением CFG. Прочитайте также главу 9, «Статический анализатор `Clang`», описывающую устройство мощного статического анализатора для `Clang`.

В нашем примере параметры `client_data` и `parent` игнорируются. Мы просто проверяем с помощью `clang_getCursorKind()`, ссылается ли текущий курсор на функцию (`CXCursor_FunctionDecl`) или метод (`CXCursor_CXXMethod`). Если выясняется, что получен курсор нужного типа, из него извлекается нужная информация: `clang_getCursorSpelling()` возвращает фрагмент кода, соответствующий данному узлу AST, и `clang_getCursorLocation()` возвращает связанный с ним объект `CXSourceLocation`. Далее мы выводим их почти так же, как в реализации проекта вывода диагностических сообщений, и завершаем функцию, возвращая из нее `CXChildVisit_Continue`. В данном случае мы уверены в отсутствии объявлений вложенных функций и потому нет смысла выполнять обход дочерних узлов.

Если курсор принадлежит другому типу, мы просто продолжаем рекурсивный обход дерева AST, возвращая `CXChildVisit_Recurse`.

Остальной код за пределами функции `visitNode` прост и понятен. Для парсинга параметров командной и содержимого файла используется уже привычный шаблонный код. Далее вызывается `visitChildren()`, которой передается курсор верхнего уровня и наша функция

обратного вызова. Последний параметр в вызове функции – указатель на клиентские данные, который в нашем примере не используется и в нем передается значение NULL.

Мы опробовали этот проект со следующим файлом:

```
#include <stdio.h>
int main() {
    printf("hello, world!");
}
```

и получили такой результат:

```
$ ./myproject hello.c

hello.c:2:5 declares main
```

Этот проект выводит также огромный объем информации, ссылаясь на каждую строку в заголовочном файле `stdio.h`, где объявляется множество функций, но мы опустили эту информацию для краткости.

Сериализация AST с помощью предварительно скомпилированных заголовочных файлов

Clang позволяет преобразовать AST в последовательную форму и сохранить в файле `.pch`, что дает возможность увеличить скорость компиляции за счет отсутствия необходимости повторно компилировать заголовочные файлы всякий раз, когда они подключаются в исходных файлах проекта. В этом случае все заголовочные файлы предварительно компилируются в общий файл PCH и затем, в ходе компиляции единицы трансляции, информация из него извлекается по мере необходимости.

Чтобы создать файл PCH для C, например, можно использовать тот же синтаксис, который применяется в GCC, заключающийся в применении флагов `-x` и `c-header`:

```
$ clang -x c-header myheader.h -o myheader.h.pch
```

Чтобы задействовать файл PCH, его следует передать вместе с флагом `-include`:

```
$ clang -include myheader.h myproject.c -o myproject
```

Семантический анализ

В ходе семантического анализа компилятор проверяет, не нарушает ли код систему типов языка, используя таблицу символов. Эта таблица хранит, кроме всего прочего, связи между идентификаторами

(символами) и их типами. На первый взгляд кажется очевидным, что проверка типов должна выполняться после парсинга, путем обхода дерева AST и сбора информации о типах символов.

Однако Clang не выполняет обход дерева AST после парсинга. Вместо этого проверка типов выполняется «на лету», в процессе создания узлов AST. Давайте вернемся к примеру парсинга `min.c`. В этом случае функция `ParseIfStatement` вызывает семантическую операцию `ActOnIfStmt` для проверки семантики инструкции `if` и выводит соответствующее диагностическое сообщение. В `lib/Parse/ParseStmt.cpp`, строка 1082, можно видеть, как выполняется передача управления механизму семантического анализа:

```
...
return Actions.ActOnIfStmt(IfLoc, FullCondExp, ...);
...
```

В помощь семантическому анализу, базовый класс `DeclContext` содержит ссылки на все узлы `Decl` в каждой области видимости, от первого до последнего. Это упрощает семантический анализ, потому что для поиска символа и проверки его типа механизму семантического анализа достаточно проверить объявления символов в узлах AST, наследующих `DeclContext`. Примерами таких узлов являются `TranslationUnitDecl`, `FunctionDecl` и `LabelDecl`.

Используя файл `min.c` в качестве примера, мы можем вывести контексты объявлений, как показано ниже:

```
$ clang -fsyntax-only -Xclang -print-decl-contexts min.c
[translation unit] 0x7faf320288f0
  <typedef> __int128_t
  <typedef> __uint128_t
  <typedef> __builtin_va_list
  [function] f(a, b)
    <parameter> a
    <parameter> b
```

Обратите внимание, что были выведены только объявления внутри `TranslationUnitDecl` и `FunctionDecl`, потому что это единственные узлы, наследующие `DeclContext`.

Проверка наличия семантических ошибок

Следующий файл `sema.c` содержит два определения идентификатора `a`:

```
int a[4];
int a[5];
```

Ошибка здесь заключается в использовании одного того же имени для определения двух разных переменных, имеющих разные типы. Эта ошибка должна выявляться на этапе семантического анализа и Clang сообщает о ней:

```
$ clang -c sema.c

sema.c:3:5: error: redefinition of 'a' with a different type

int a[5];
    ^

sema.c:2:5: note: previous definition is here

int a[4];
    ^

1 error generated.
```

Если проверить этот файл с помощью нашего диагностического проекта, он выведет следующее:

```
$ ./myproject sema.c

Severity: 3 File: sema.c Line: 2 Col: 5 Category: "Semantic Issue"
Message: redefinition of 'a' with a different type:
'int [5]' vs 'int [4]'
```

Создание кода промежуточного представления LLVM IR

После выполнения парсинга в сочетании с семантическим анализом, функция `ParseAST` вызывает метод `HandleTranslationUnit`, чтобы передать клиенту окончательное дерево `AST`. Если драйвер компилятора использовал операцию `CodeGenAction`, таким клиентом будет `BackendConsumer`, выполняющий обход дерева `AST` и генерирующий код LLVM IR, который реализует поведение, описываемое данным деревом. Преобразование в LLVM IR начинается с объявления верхнего уровня `TranslationUnitDecl`.

Если продолжить использовать в качестве примера файл `min.c`, инструкция `if` будет преобразована в код LLVM IR в файле `lib/CodeGen/CGStmt.cpp`, в строке 130, функцией `EmitIfStmt`. Воспользуемся отладчиком и посмотрим цепочку вызовов от `ParseAST` до `EmitIfStmt`:

```
$ gdb clang
(gdb) b CGStmt.cpp:130
```

```
(gdb) r -ccl -emit-obj min.c
...
130 case Stmt::IfStmtClass: EmitIfStmt(cast<IfStmt>(*S));
break;
(gdb) backtrace
#0 clang::CodeGen::CodeGenFunction::EmitStmt
#1 clang::CodeGen::CodeGenFunction::EmitCompoundStmtWithoutScope
#2 clang::CodeGen::CodeGenFunction::EmitFunctionBody
#3 clang::CodeGen::CodeGenFunction::GenerateCode
#4 clang::CodeGen::CodeGenModule::EmitGlobalFunctionDefinition
#5 clang::CodeGen::CodeGenModule::EmitGlobalDefinition
#6 clang::CodeGen::CodeGenModule::EmitGlobal
#7 clang::CodeGen::CodeGenModule::EmitTopLevelDecl
#8 (anonymous namespace)::CodeGeneratorImpl::HandleTopLevelDecl
#9 clang::BackendConsumer::HandleTopLevelDecl
#10 clang::ParseAST
```

Преобразованием в код LLVM IR мы завершаем наш тур по анализатору исходного кода. Далее, в процессе компиляции, обычно выполняется оптимизация кода LLVM IR и генератор выполняемого кода создает двоичный код для целевой архитектуры. Если у вас появится желание реализовать собственный анализатор исходного кода для своего языка программирования, мы рекомендуем обратиться к руководству «Kaleidoscope» (<http://llvm.org/docs/tutorial>). В следующем разделе мы покажем, как написать упрощенный драйвер Clang, объединяющий в себе все этапы, обсуждавшиеся выше.

Все вместе

В этом примере мы воспользуемся возможностью познакомить вас с C++-интерфейсом Clang и не будем использовать C-интерфейс libclang. Здесь мы напишем программу, выполняющую лексический, синтаксический и семантический анализ, используя для этого внутренние классы Clang; то есть, у нас будет возможность реализовать простой объект FrontendAction. Вы можете продолжать использовать Makefile, что приводился в начале главы. Однако мы рекомендуем убрать флаги компилятора `-Wall -Wextra`, потому что они производят слишком много предупреждений для заголовочных файлов Clang, касающихся неиспользуемых параметров.

Исходный код примера представлен ниже:

```
#include "llvm/ADT/IntrusiveRefCntPtr.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Host.h"
#include "clang/AST/ASTContext.h"
```

```
#include "clang/AST/ASTConsumer.h"
#include "clang/Basic/Diagnostic.h"
#include "clang/Basic/DiagnosticOptions.h"
#include "clang/Basic/FileManager.h"
#include "clang/Basic/SourceManager.h"
#include "clang/Basic/LangOptions.h"
#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/TargetOptions.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/TextDiagnosticPrinter.h"
#include "clang/Lex/Preprocessor.h"
#include "clang/Parse/Parser.h"
#include "clang/Parse/ParseAST.h"
#include <iostream>

using namespace llvm;
using namespace clang;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"), cl::Required);

int main(int argc, char **argv)
{
    cl::ParseCommandLineOptions(argc, argv, "My simple front end\n");
    CompilerInstance CI;
    DiagnosticOptions diagnosticOptions;
    CI.createDiagnostics();

    IntrusiveRefCntPtr<TargetOptions> PTO(new TargetOptions());
    PTO->Triple = sys::getDefaultTargetTriple();
    TargetInfo *PTI = TargetInfo::CreateTargetInfo(CI.getDiagnostics(),
                                                  PTO.getPtr());

    CI.setTarget(PTI);
    CI.createFileManager();
    CI.createSourceManager(CI.getFileManager());
    CI.createPreprocessor();
    CI.getPreprocessorOpts().UsePredefines = false;
    ASTConsumer *astConsumer = CreateASTPrinter(NULL, "");
    CI.setASTConsumer(astConsumer);

    CI.createASTContext();
    CI.createSema(TU_Complete, NULL);
    const FileEntry *pFile = CI.getFileManager().getFile(FileName);
    if (!pFile) {
        std::cerr << "File not found: " << FileName << std::endl;
        return 1;
    }
    CI.getSourceManager().createMainFileID(pFile);
    CI.getDiagnosticsClient().BeginSourceFile(CI.getLangOpts(), 0);
```

```

ParseAST(CI.getSema());
// Вывод статистической информации о дереве AST
CI.getASTContext().PrintStats();
CI.getASTContext().Idents.PrintStats();
return 0;
}

```

Предыдущий код выполняет лексический, синтаксический и семантический анализ исходного файла, указанного в виде аргумента командной строки. Работа примера завершается выводом статистической информации об исходном коде и дереве AST. Представленная программа выполняет следующие шаги:

1. Управление всей инфраструктурой компиляции осуществляет класс `CompilerInstance` (см. http://clang.llvm.org/doxygen/classclang_1_1CompilerInstance.html). Поэтому первым делом программа создает экземпляр этого класса и сохраняет его в переменной `CI`.
2. Обычно инструмент `clang -ccl` создает определенный экземпляр `FrontendAction`, выполняющий все шаги, описанные в этой главе. Так как мы хотели бы показать вам, как эти шаги выполняются в действительности, мы не будем использовать `FrontendAction`, а будем использовать `CompilerInstance` вручную. Мы использовали метод `CompilerInstance::createDiagnostics` для создания механизма диагностики и установили текущую цель, взяв ее из системы.
3. Далее создаются три новых ресурса: диспетчер файлов, диспетчер источников данных и препроцессор. Первый необходим для чтения исходных данных, а второй отвечает за управление экземплярами `SourceLocation`, которые используются в ходе лексического анализа и парсинга.
4. На следующем шаге создается ссылка на потребителя `ASTConsumer` и сохраняется в объекте `CI`. Таким способом внешний клиент может получить окончательное дерево AST (после парсинга и семантического анализа). Например, если бы нам потребовалось, чтобы этот драйвер сгенерировал код LLVM IR, мы могли бы передать ссылку на определенную реализацию `ASTConsumer` (называется `BackendConsumer`). Именно так `CodeGenAction` устанавливает ссылку `ASTConsumer` в своем экземпляре `CompilerInstance`. В данном примере подключается заголовочный файл `ASTConsumers.h` с объявлениями нескольких видов потребителей, и мы использовали потребителя, который просто выводит дерево AST в консоль. Созда-

ется такой потребитель вызовом `CreateASTPrinter()`. Если вам будет интересно, потратьте некоторое время на реализацию собственного класса потребителя, наследующего `ASTConsumer`, который выполнял бы какой-нибудь анализ (начните с изучения файла `lib/Frontend/ASTConsumers.cpp`, где приводится несколько примеров реализации подобных классов).

5. Затем создается новый объект `ASTContext` для парсера, объект `Sema` для механизма семантического анализа и оба объекта добавляются в объект `SI`. Попутно инициализируется приемник диагностических сообщений (в данном случае – стандартный приемник, который просто выводит сообщения на экран).
6. Далее вызывается `ParseAST` для выполнения лексического и синтаксического анализа, в ходе которого посредством функции `HandleTranslationUnit` будет вызываться `ASTConsumer`. При обнаружении серьезной ошибки на любом этапе, Clang также выведет диагностическое сообщение и прервет работу конвейера.
7. В заключение выводится статистическая информация.

Проверим получившийся инструмент со следующим файлом:

```
int main() {
    char *msg = "Hello, world!\n";
    write(1, msg, 14);
    return 0;
}
```

Ниже приводятся полученные результаты:

```
$ ./myproject test.c
```

```
int main() {
    char *msg = "Hello, world!\n";
    write(1, msg, 14);
    return 0;
}
```

```
*** AST Context Stats:
```

```
39 types total.
 31 Builtin types
  3 Complex types
  3 Pointer types
  1 ConstantArray types
  1 FunctionNoProto types
```

```
Total bytes = 544
```

```
0/0 implicit default constructors created
```

```
0/0 implicit copy constructors created
```

```
0/0 implicit copy assignment operators created
```

```
0/0 implicit destructors created
```

```
Number of memory regions: 1
```

```
Bytes used: 1594
```

```
Bytes allocated: 4096
```

```
Bytes wastes: 2502 (includes alignment, etc)
```

В заключение

В этой главе мы описали особенности работы анализатора исходного кода для Clang. Мы объяснили разницу между библиотеками анализатора исходного кода Clang, драйвером компилятора и фактическим компилятором `clang -cc1`. Мы также поговорили о диагностике и показали небольшую программу на основе библиотеки `libclang`, которая выводит диагностические сообщения. Затем мы прошли по всем этапам работы анализатора исходного кода: лексический, синтаксический и семантический анализ, и создание промежуточного представления, показав, как Clang реализует эти этапы. В заключение был представлен пример реализации простого драйвера компилятора. Желаям поближе с AST мы рекомендуем обратиться к документу <http://clang.llvm.org/docs/IntroductionToTheClangAST.html>. Если вам интересно будет лучше узнать архитектуру Clang, обязательно посетите страницу <http://clang.llvm.org/docs/InternalsManual.html>.

В следующей главе мы сделаем еще один шаг вперед и познакомимся с промежуточным представлением LLVM.



ГЛАВА 5.

Промежуточное представление LLVM

Промежуточное представление (Intermediate Representation, IR) LLVM – это магистраль, связывающая анализаторы исходного кода и генераторы выполняемого кода, которая позволяет LLVM анализировать программы на самых разных языках программирования и генерировать код для разных целевых архитектур. Анализаторы исходного кода производят IR, а генераторы выполняемого кода потребляют его. Промежуточное представление IR также является точкой применения архитектурно-независимых оптимизаций. В этой главе мы охватим следующие темы:

- ♦ характеристики LLVM IR;
- ♦ синтаксис языка LLVM IR;
- ♦ как реализовать генератор LLVM IR;
- ♦ структура проходов LLVM IR;
- ♦ как реализовать свой проход IR.

Обзор

Выбор компилятора IR является очень ответственным решением. Он определяет, какие оптимизации смогут применяться для ускорения выполнения кода. С одной стороны высокоуровневое промежуточное представление позволяет оптимизаторам легко определять намерения исходного кода. С другой стороны низкоуровневое промежуточное представление упрощает для компилятора задачу создания машинного кода. Чем больше информации имеется о целевой аппаратной архитектуре, тем шире возможности исследования ее особенностей. Кроме того, чем ниже уровень промежуточного представления, тем больше осторожности оно требует. Чем ближе промежуточное представление к машинным инструкциям, тем сложнее компилятору отображать

фрагменты программы в оригинальный исходный код. Более того, если компилятор окажется связан с какой-то целевой архитектурой особенно тесно, ему будет сложно генерировать выполняемый код для других архитектур, имеющих свои отличительные особенности.

Необходимость чем-то жертвовать привела к выбору разных решений в разных компиляторах. В некоторых компиляторах, например, поддерживается создание выполняемого кода только для какой-то одной целевой архитектуры. Это позволяет использовать специализированные промежуточные представления IR, обеспечивающие более высокую эффективность компиляции. Примером таких компиляторов может служить Intel C++ Compiler (`icc`). Однако, создание компилятора, поддерживающего какую-то одну архитектуру, является слишком дорогим решением, если имеется необходимость поддерживать несколько архитектур. В таких случаях почти невозможно написать несколько компиляторов, по одному для каждой архитектуры, и лучше написать один компилятор, поддерживающий несколько архитектур. Именно такой подход был реализован в GCC и LLVM.

В этих проектах, которые называют *перенастраиваемыми компиляторами* (*retargetable compilers*), приходится решать значительно более широкий спектр проблем, связанных с координацией создания выполняемого кода для разных платформ. Ключом к минимизации усилий, которые необходимо приложить для создания перенастраиваемого компилятора, является поддержка промежуточного представления IR, которое могут использовать генераторы выполняемого кода для разных целевых архитектур. Имея общее промежуточное представление, можно реализовать комплекс оптимизаций, независимых от поддерживаемых платформ, но это требует от разработчиков поднять уровень IR, чтобы обеспечить максимальную универсальность. Поскольку работа на более высоких уровнях препятствует применению некоторых приемов, характерных для какой-то одной платформы, хорошие перенастраиваемые компиляторы используют несколько промежуточных представлений более низкого уровня для применения дополнительных оптимизаций.

Проект LLVM начинался с создания IR, действующего на более низком уровне, чем байткод Java, о чем свидетельствует первоначальная расшифровка аббревиатуры LLVM – Low Level Virtual Machine (низкоуровневая виртуальная машина). Основной идеей проекта было исследование приемов низкоуровневой оптимизации и оптимизации во время компоновки. Оптимизации времени компоновки возможны при сохранении IR на диск, как и при использовании байт-

кода. Байткод даст пользователю возможность объединить несколько модулей в один файл и затем применить межпроцедурные оптимизации. То есть оптимизации воздействуют на множество единиц компиляции, как если бы они находились в одном модуле.

В главе 3, «Инструменты и организация», говорилось, что в настоящее время проект LLVM не является ни виртуальной машиной, ни конкурентом Java и реализует собственное промежуточное представление, обеспечивающее высокую эффективность. Например, помимо LLVM IR – типичного промежуточного представления, к которому могут применяться универсальные приемы оптимизации – каждый генератор выполняемого кода может выполнять собственные оптимизации, характерные для той или иной аппаратной платформы, когда программа представлена классами `MachineFunction` и `MachineInstr`. Эти классы представляют программу с использованием машинных инструкций.

С другой стороны, не менее важными являются классы `Function` и `Instruction`, образующие универсальное промежуточное представление IR, на основе которого может создаваться машинный код. Это промежуточное представление почти не зависит от конкретных особенностей целевых архитектур (за редким исключением) и считается *официальным* промежуточным представлением LLVM. Чтобы избежать путаницы с другими уровнями представления программ в LLVM, которые, технически также могут считаться промежуточными представлениями, мы не будем использовать это название по отношению к ним и зарезервируем его для обозначения официального, универсального промежуточного представления, которое образуется классом `Instruction` и некоторыми другими. Такой же подход к терминологии принят в документации LLVM.

Проект LLVM начинался как множество инструментов, окружающих LLVM IR, что объясняет количество и зрелость оптимизаторов, действующих на этом уровне. Промежуточное представление IR имеет три эквивалентные формы:

- представление в памяти (класс `Instruction` и другие);
- представление на диске в экономичном формате (файлы биткода);
- представление на диске в удобочитаемом текстовом формате (файлы сборок LLVM).

В LLVM имеются инструменты и библиотеки, позволяющие работать со всеми формами промежуточного представления. Соответственно, эти инструменты способны преобразовывать промежуточ-

ное представление из одной формы в другую и применять различные оптимизации, как показано на рис. 5.1.

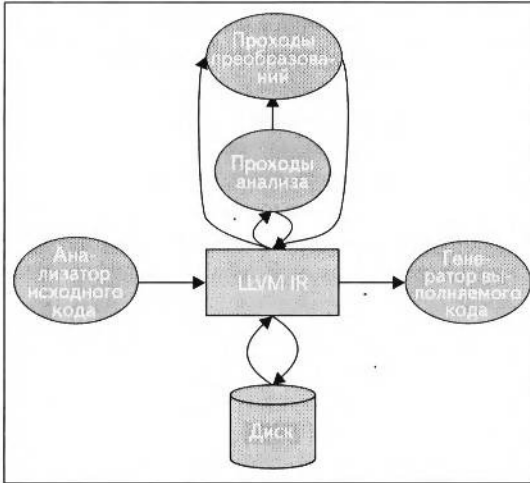


Рис. 5.1. Инструменты LLVM способны преобразовывать промежуточное представление из одной формы в другую

Зависимость LLVM IR от целевой архитектуры

Разработчики стремились сделать промежуточное представление LLVM IR максимально независимым от целевой архитектуры, но в нем еще остаются некоторые специфические черты. Многие критикуют язык C/C++ за его врожденную зависимость от платформы. Например, когда программа подключает стандартные заголовочные файлы C в системе Linux, она неявно подключает некоторые заголовочные файлы из каталога `bits`. В этом каталоге хранятся файлы для конкретной целевой архитектуры с определениями типов, соответствующих данным, которые ожидаются **системными вызовами** в ядре. Когда анализатор исходного кода выполняет парсинг, он также должен учитывать, что на разных архитектурах тип `int` имеет разные размеры.

Как результат, и библиотечные заголовочные файлы, и типы данных в языке C уже зависят от целевой архитектуры, что усложняет задачу создания промежуточного представления программы, которое



позднее может быть оттранслировано в выполняемый код для разных архитектур. Если программа пишется с использованием заголовочных файлов стандартной библиотеки C, дерево AST такой программы всегда будет получаться зависимым от целевой архитектуры, еще до преобразования ее в промежуточное представление. Кроме того, анализатор исходного кода генерирует IR, используя определенные размеры типов, соглашения о вызовах и специальные библиотечные функции, соответствующие требованиям целевого двоячного интерфейса приложений (Application Binary Interface, ABI). И все же, представление LLVM IR достаточно универсально, чтобы справиться с архитектурными различиями достаточно абстрактным способом.

Основные инструменты для работы с форматами IR

Выше уже упоминалось, что промежуточное представление LLVM IR может храниться на диске в двух форматах: в виде двоичного биткода и текстовых сборок. В этом разделе мы посмотрим, как можно использовать их. В качестве основы будем использовать представленный ниже исходный код в файле `sum.c`:

```
int sum(int a, int b) {  
    return a+b;  
}
```

Сгенерировать соответствующий этой программе биткод можно командой:

```
$ clang sum.c -emit-llvm -c -o sum.bc
```

Файл сборки можно получить командой:

```
$ clang sum.c -emit-llvm -S -c -o sum.ll
```

Содержимое файла сборки LLVM IR можно преобразовать в биткод командой:

```
$ llvm-as sum.ll -o sum.bc
```

Обратное преобразование – биткода в файл сборки – производится с помощью дизассемблера:

```
$ llvm-dis sum.bc -o sum.ll
```

Инструмент `llvm-extract` позволяет извлекать функции IR и глобальные переменные, а также удалять глобальные переменные из

модуля IR. Например, извлечь функцию `sum` из файла `sum.bc` можно следующей командой:

```
$ llvm-extract -func=sum sum.bc -o sum-fn.bc
```

В данном конкретном примере вы не заметите никаких различий между файлами `sum.bc` и `sum-fn.bc`, потому что функция `sum` является единственной конструкцией в этом модуле.

Введение в синтаксис языка LLVM IR

Рассмотрим файл сборки LLVM IR `sum.ll`:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
  i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-
  a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.7.0"

define i32 @sum(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  %1 = load i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}

attributes #0 = { nounwind ssp uwtable ... }
```

Содержимое файла LLVM, текстовой сборки или двоичного биткода, определяет модуль LLVM. Модуль – это структура данных LLVM IR верхнего уровня. Каждый модуль содержит последовательность функций, каждая из которых состоит из группы базовых блоков, которые в свою очередь содержат последовательности инструкций. Модуль также включает также некоторые дополнительные объекты поддержки этой модели, такие как глобальные переменные, определение целевого формата данных, прототипы внешних функций и объявления структур данных.

Локальные значения LLVM являются аналогами регистров в языке ассемблера и могут иметь любые имена, начинающиеся с символа `%`. То есть, инструкция `%add = add nsw i32 %0, %1` складывает ло-

кальные значения %0 и %1, и сохраняет результат в новом локальном значении %add. Вы можете выбирать любые имена для значений, но, если вы предпочитаете краткость изобретательности, можете использовать простые числа. В этом коротком примере уже можно увидеть некоторые фундаментальные особенности LLVM:

Использование формы **Static Single Assignment (SSA)**. Обратите внимание, что ни одному значению не выполняется повторное присваивание – присваивание каждому из них выполняется только один раз, в момент определения. Каждую операцию с участием значения можно отследить назад до определения этого значения. Это значительно упрощает оптимизацию, благодаря простоте цепочек использование/определение, которые создает форма SSA. Если бы в LLVM не использовалась форма SSA, могло бы потребоваться выполнять отдельный анализ потоков данных для поиска цепочек использование/определение, которые совершенно необходимы для классических оптимизаций, таких как распространение констант (constant propagation) и удаление общих подвыражений (common subexpression elimination).

Код организован как последовательность трехадресных инструкций. Инструкции обработки данных получают два исходных операнда и сохраняют результат в отдельном, третьем операнде.

Имеется бесконечное множество регистров. Обратите внимание, что локальные значения могут иметь любые имена, начинающиеся с символа %, в том числе и числовые, начиная с нуля, такие как %0, %1, и отсутствуют какие-либо ограничения на число различных значений.

Конструкция `target datalayout` содержит информацию о порядке следования байтов (endianness) и размерах типов для триады `target triple`, описывающей целевой хост. Некоторые виды оптимизации зависят от конкретного формата данных. Давайте поближе рассмотрим объявление формата данных:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-  
i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-  
a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"  
target triple = "x86_64-apple-macosx10.7.0"
```

Из этой строки вытекают следующие факты:

- Целевой платформой является macOS 10.7.0 на аппаратной платформе x86_64. Данные в памяти хранятся в формате с обратным порядком следования байтов (little-endian), о чем сообщает первая буква в определении (буква «e» нижнего регистра).

На архитектурах с прямым порядком следования байтов (big-endian) первой будет следовать буква «E» верхнего регистра.

- Информация о типах имеет формат: `type:<size>:<abi>:<preferred>`. Определение `p:64:64:64` в предыдущем примере представляет указатель, имеющий размер 64 бита, с выравниванием `abi` и `preferred` по границам 64 бит. Выравнивание `abi` определяет минимально необходимое выравнивание для описываемого типа, а выравнивание `preferred` – наибольшее возможное выравнивание, которое может принести дополнительные выгоды. 32-разрядные целочисленные типы `i32:32:32` имеют размер, равный 32 битам, с выравниванием `abi` и `preferred` по границам 32 бит, и так далее.

Определение функции очень похоже на определение функции в языке C:

```
define i32 @sum(i32 %a, i32 %b) #0 {
```

Эта функция возвращает значение типа `i32` и принимает два аргумента типа `i32`, `%a` и `%b`. Локальные идентификаторы всегда должны начинаться с символа `%`, а глобальные – с символа `@`. LLVM поддерживает широкий спектр типов данных, но наиболее важными являются следующие:

- целые числа произвольного размера в форме `iN`, например: `i32`, `i64` и `i128`;
- вещественные числа, такие как 32-разрядные одинарной точности (`float`) и 64-разрядные двойной точности (`double`);
- векторные типы, определения которых имеют вид: `<<# elements> x <elementtype>>`; тип, соответствующий вектору с четырьмя элементами типа `i32` определяется как: `<4 x i32>`.

Тег `#0` в определении функции отображается в множество атрибутов функции, также близко напоминающее множество атрибутов функций и методов в языке C/C++. Собственно множество атрибутов определяется в конце файла:

```
attributes #0 = { nounwind ssp uwtable
  "less-precise-fpmad"="false"
  "no-frame-pointer-elim"="true"
  "no-frame-pointer-elim-non-leaf"="true"
  "no-infs-fp-math"="false"
  "no-nans-fp-math"="false"
  "unsafe-fp-math"="false"
  "use-softfloat"="false" }
```

Например, атрибутом `nounwind` отмечаются функции и методы, которые не возбуждают исключений. Атрибут `ssp` сообщает генератору кода, что он должен использовать защиту от срыва стека (`stack smash protection`), чтобы улучшить стойкость кода к атакам.

Тело функции явно разделено на **базовые блоки (Basic Blocks, BB)** и каждый новый блок начинается с метки. Метка связана с базовым блоком точно так же, как идентификатор значения с инструкцией. Если метка опущена, ассемблер LLVM сгенерирует ее, опираясь на собственные соглашения об именовании. Базовый блок – это последовательность инструкций с единственной точкой входа в первой инструкции и единственной точкой выхода в последней инструкции. То есть, когда код выполняет переход к метке, соответствующей базовому блоку, можно быть уверенными, что будут выполнены все инструкции в этом блоке, от первой до последней, причем последняя инструкция направит поток выполнения к другому блоку. Базовые блоки и соответствующие им метки должны соответствовать следующим условиям:

- каждый базовый блок должен завершаться инструкцией, выполняющей переход к следующему базовому блоку или возврат из функции;
- первый базовый блок, который еще называют блоком входа, – играет особую роль в функциях LLVM и не должен использоваться как цель ни в каких инструкциях ветвления.

Наш файл LLVM, `sum.ll`, имеет только один базовый блок, потому что в программе отсутствуют переходы, циклы или вызовы функций. Начало функции отмечено меткой `entry`, а в конце находится инструкция возврата `ret`:

```
entry:  
  %a.addr = alloca i32, align 4  
  %b.addr = alloca i32, align 4  
  store i32 %a, i32* %a.addr, align 4  
  store i32 %b, i32* %b.addr, align 4  
  %0 = load i32* %a.addr, align 4  
  %1 = load i32* %b.addr, align 4  
  %add = add nsw i32 %0, %1  
  ret i32 %add
```

Инструкция `alloca` резервирует пространство в кадре стека текущей функции. Объем пространства определяется размером типа элемента с учетом указанного выравнивания. Первая инструкция, `%a.addr = alloca i32, align 4`, выделяет место на стеке для 4-байтного элемента с выравниванием по границе 4 байт. Указатель

на элемент в стеке сохраняется в локальном идентификаторе `%a.addr`. Инструкция `alloca` обычно используется для представления локальных (автоматических) переменных.

Аргументы `%a` и `%b` сохраняются на стеке, по адресам `%a.addr` и `%b.addr` с помощью инструкций `store`. Загрузка значений выполняется инструкциями `load`, далее эти значения используются в операции сложения, `%add = add nsw i32 %0, %1`. Наконец, результат сложения, `%add`, возвращается из функции. Флаг `nsw` указывает, что данная операция сложения не должна проверять перенос в знаковый бит (`no signed wrap`), что обеспечивает возможность дополнительной оптимизации. Если вам интересно узнать предысторию флага `nsw`, прочитайте статью Дэна Гохмана (Dan Gohman) на LLVMdev: <http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html>.

Инструкции `load` и `store` фактически являются избыточными – аргументы функции можно использовать в инструкции `add` непосредственно. Clang использует по умолчанию уровень оптимизации `-O0` (нет оптимизации), поэтому избыточные инструкции `load` и `store` не удаляются. Если скомпилировать с флагом оптимизации `-O1`, на выходе получится более простой код:

```
define i32 @sum(i32 %a, i32 %b) ... {
entry:
    %add = add nsw i32 %b, %a
    ret i32 %add
}
...
```

Сборки LLVM очень удобно использовать для создания небольших примеров с целью тестирования генераторов выполняемого кода и изучения основ LLVM. Однако разработчикам анализаторов исходного кода для создания LLVM IR рекомендуется использовать библиотечный интерфейс, о котором рассказывается в следующем разделе. Полный справочник по синтаксису сборок LLVM IR можно найти по адресу: <http://llvm.org/docs/LangRef.html>.

Представление LLVM IR в памяти

Представление LLVM IR в памяти близко моделирует синтаксис языка LLVM, с которым мы только что познакомились. Заголовочные файлы с определениями классов C++ для хранения IR в памяти находятся в каталоге `include/llvm/IR`. Далее перечислены наиболее важные классы.

- Класс `Module` объединяет все данные, используемые в единице трансляции, и является аналогом «модуля» в терминологии LLVM. Объявляет тип итератора `Module::iterator`, обеспечивающий простой способ обхода функций внутри модуля. Получить итератор можно с помощью методов `begin()` и `end()`. Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1Module.html.
- Класс `Function` содержит все объекты, связанные с определением или объявлением функции. В случае объявления (проверить, представляет ли объект `Function` объявление функции, можно с помощью метода `isDeclaration()`) он содержит только прототип функции. В обоих случаях он содержит список параметров, доступный через метод `getArgumentList()` или пару методов `arg_begin()` и `arg_end()`. Выполнить обход параметров можно с помощью итератора `Function::arg_iterator`. Если объект `Function` представляет определение функции, имеется возможность выполнить составляющие ее базовые блоки, используя идиому: `Function::iterator i = function.begin(), e = function.end(); i != e; ++i`. Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html.
- Класс `BasicBlock` инкапсулирует последовательность инструкций LLVM, доступную с использованием идиомы `begin()/end()`. С помощью метода `getTerminator()` можно обратиться непосредственно к последней инструкции. Имеется также несколько вспомогательных методов для навигации, такие как `getSinglePredecessor()`, для доступа к предшествующему базовому блоку, если текущий базовый блок имеет единственный предшествующий блок. В случае нескольких предшествующих блоков вам придется самостоятельно выполнить обход списка, для чего, впрочем, достаточно обойти все базовые блоки и проверить завершающие их инструкции. Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html.
- Класс `Instruction` представляет элементарную единицу компиляции в LLVM IR – единственную инструкцию. Имеет несколько методов-предикатов высокого уровня, таких как `isAssociative()`, `isCommutative()`, `isIdempotent()` и `isTerminator()`, но точное значение инструкции можно получить вызовом метода `getOpcode()`, который возвращает эле-

мент перечисления `llvm::Instruction`, представляющего коды операций LLVM IR. Получить доступ к операндам можно с помощью пары методов `op_begin()` и `op_end()`, унаследованных от класса `User` (см. ниже). Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html.

Мы еще не познакомили вас с наиболее мощными аспектами LLVM IR (которые обеспечивает форма SSA): интерфейсами `Value` и `User`. Они позволяют легко перемещаться от определений к случаям использования и обратно. Класс, наследующий `Value`, определяет результат, который может использоваться где-то еще, а класс, наследующий `User`, определяет сущность, использующую один или более интерфейсов `Value`. Классы `Function` и `Instruction`, оба являются подклассами `Value` и `User`, а класс `BasicBlock` наследует только `Value`. Чтобы проще было понять суть, ниже приводится более подробное описание классов.

- Класс `Value` определяет методы `use_begin()` и `use_end()`, позволяющие выполнять итерации по экземплярам `User` и упрощающие следование по цепочкам от определения до использования. Каждый экземпляр класса `Value` имеет также метод `getName()`, дающий доступ к имени значения. Он учитывает тот факт, что любое значение в LLVM может иметь уникальный идентификатор. Например, имя `%add1` может идентифицировать результат инструкции `add`, имя `bb1` может идентифицировать базовый блок, а имя `myfunc` – функцию. Класс `Value` имеет также мощный метод `replaceAllUsesWith(Value *)` для навигации по всем пользователям (экземплярам `User`) данного значения и замещения его некоторым другим значением. Это отличный пример, как форма SSA позволяет легко замещать инструкции и выполнять оптимизации. Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1Value.html.
- Класс `User` имеет методы `op_begin()` и `op_end()` для быстрого доступа ко всем используемым интерфейсам `Value`. Обратите внимание, что таким способом реализуется модель цепочки от использования до определения. Имеется также вспомогательный метод `replaceAllUsesOfWith(Value *From, Value *To)` для замены любого из используемых значений. Полный интерфейс класса приводится на странице: http://llvm.org/docs/doxygen/html/classllvm_1_1User.html.

Реализация собственного генератора LLVM IR

Имеется возможность использовать API генератора LLVM IR, чтобы программно создать промежуточное представление `sum.ll` (с уровнем оптимизации `-O0`, то есть, без оптимизации). А этом разделе вы увидите, как сделать это, шаг за шагом. Для начала посмотрим, какие заголовочные файлы понадобятся.

- `#include <llvm/ADT/SmallVector.h>`: содержит определение шаблона `SmallVector<>` структуры данных, упрощающей создание эффективных векторов, когда число элементов невелико. Справку по структурам данных LLVM можно получить по адресу: <http://llvm.org/docs/ProgrammersManual.html>.
- `#include <llvm/Analysis/Verifier.h>`: содержит определение прохода проверки (`verifier pass`), выполняющего важный анализ, в ходе которого проверяется, насколько ваш модуль LLVM соответствует правилам оформления IR.
- `#include <llvm/IR/BasicBlock.h>`: содержит объявление класса базового блока `BasicBlock`, важнейшего компонента IR, который уже был представлен выше.
- `#include <llvm/IR/CallingConv.h>`: определяет набор правил ABI, используемых для вызова функций, такие как место хранения аргументов функций.
- `#include <llvm/IR/Function.h>`: содержит объявление класса `Function`, компонента IR.
- `#include <llvm/IR/Instructions.h>`: содержит объявления всех подклассов класса `Instruction`, фундаментальной структуры данных IR.
- `#include <llvm/IR/LLVMContext.h>`: хранит определения глобальных данных библиотеки LLVM, которые позволяют создавать многопоточные реализации для работы с разными контекстами в разных потоках.
- `#include <llvm/IR/Module.h>`: содержит объявление класса `Module`, элемента верхнего уровня в иерархии IR.
- `#include <llvm/Bitcode/ReaderWriter.h>`: содержит объявления инструментов для чтения/записи файлов с биткодом LLVM.
- `#include <llvm/Support/ToolOutputFile.h>`: содержит объявление вспомогательного класса, используемого для записи в выходной файл.

В данном примере потребуется также импортировать символы из пространства имен `llvm`:

```
using namespace llvm;
```

Теперь приступим к программному коду. Напишем его в несколько этапов:

Прежде всего напомним новую вспомогательную функцию `makeLLVMModule`, возвращающую указатель на экземпляр `Module`, элемента IR верхнего уровня, содержащего все остальные объекты IR:

```
Module *makeLLVMModule() {
    Module *mod = new Module("sum.ll", getGlobalContext());

    mod->setDataLayout("e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
        i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-
        v128:128:128-a0:0:64-s0:64:64-f80:128:128-
        n8:16:32:64-S128");
    mod->setTargetTriple("x86_64-apple-macosx10.7.0");
}
```

Добавляя в модуль триаду (`triple`) определения архитектуры и описание формата представления данных, мы разрешаем выполнение оптимизаций, зависящих от этой информации, но при этом они должны совпадать со строками триады и описания формата представления данных, используемых в генераторе выполняемого кода. Однако, вы можете опустить эти определения, если возможность подобных аппаратно-зависимых оптимизаций вас не волнует и предполагается, что целевая архитектура будет указываться генератору выполняемого кода явно. Чтобы создать модуль, мы получаем текущий контекст LLVM из `getGlobalContext()` и определяем имя модуля. В данном случае роль имени модуля будет играть имя файла, используемого как модель, `sum.ll`, но вы можете выбрать любое другое имя. Контекст – это экземпляр класса `LLVMContext`, который служит гарантией безопасности в ситуациях, когда IR генерируется в многопоточной среде, причем каждый поток должен использовать только один контекст. Функции `setDataLayout()` и `setTargetTriple()` связывают с модулем строки триады определения архитектуры и описания формата представления данных.

1. Чтобы объявить функцию `sum`, сначала следует определить ее сигнатуру:

```
SmallVector<Type*, 2> FuncTyArgs;
FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
FunctionType *FuncTy = FunctionType::get(
```

```
/*Result=*/ IntegerType::get(mod->getContext(), 32),
/*Params=*/ FuncTyArgs, /*isVarArg=*/ false);
```

Объект `FunctionType` в данном случае определяет функцию, возвращающую 32-разрядное целое со знаком, не имеющую переменных аргументов и принимающую два 32-разрядных целочисленных аргумента.

2. Создается функция вызовом статического метода `Function::Create()`, которому передается объект типа функции `FuncTy`, созданный выше, тип компоновки и экземпляр модуля. Элемент перечисления `GlobalValue::ExternalLinkage` сообщает, что функция может быть доступна из других модулей (единиц трансляции):

```
Function *funcSum = Function::Create(
    /*Type=*/ FuncTy,
    /*Linkage=*/ GlobalValue::ExternalLinkage,
    /*Name=*/ "sum", mod);
funcSum->setCallingConv(CallingConv::C);
```

3. Далее необходимо сохранить указатели на объекты `Value`, представляющие аргументы, чтобы их можно было использовать позже. Воспользуемся для этого итератором по аргументам функции. Указатели `int32_a` и `int32_b` ссылаются на первый и второй аргументы функции, соответственно. Мы также определяем имена для обоих аргументов, что не является обязательным, потому что LLVM может подставить временные имена:

```
Function::arg_iterator args = funcSum->arg_begin();
Value *int32_a = args++;
int32_a->setName("a");
Chapter 5
[ 117 ]
Value *int32_b = args++;
int32_b->setName("b");
```

4. Чтобы начать конструировать тело функции, создадим первый базовый блок с меткой (с именем значения) `entry` и сохраним указатель на него в `labelEntry`. Мы должны также передать ссылку на функцию, которую будет содержать данный базовый блок:

```
BasicBlock *labelEntry = BasicBlock::Create(mod->getContext(),
    "entry", funcSum, 0);
```

5. Теперь базовый блок `entry` готов к заполнению инструкциями. Добавим в него две инструкции `alloca`, создав два 32-раз-

рядных элемента стека с выравниванием по границе 4 байт. В конструктор инструкции следует передать ссылку на базовый блок, куда эта инструкция помещается. По умолчанию новые инструкции вставляются в конец базового блока:

```
// Вход в блок (метка entry)
AllocaInst *ptrA = new AllocaInst(
    IntegerType::get(mod->getContext(), 32),
    "a.addr", labelEntry);
ptrA->setAlignment(4);
AllocaInst *ptrB = new AllocaInst(
    IntegerType::get(mod->getContext(), 32),
    "b.addr", labelEntry);
ptrB->setAlignment(4);
```

Совет. Для создания инструкций IR можно также использовать вспомогательный шаблонный класс `IRBuilder<>` (см. http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html). Однако мы предпочли не использовать его, чтобы показать оригинальный интерфейс. Если вы пожелаете использовать вспомогательный класс, просто подключите заголовочный файл `llvm/IR/IRBuilder.h`, создайте экземпляр класса, указав объект контекста LLVM, и вызовите метод `SetInsertPoint()`, чтобы указать, куда должны помещаться новые инструкции. Затем просто вызывайте методы создания инструкций, такие как `CreateAlloca()`.

- Сохраним аргументы `int32_a` и `int32_b` функции на стеке, используя указатели, возвращаемые инструкциями `alloca`, `ptrA` и `ptrB`. Хотя указатели на инструкции `store` сохраняются в значениях `st0` и `st1`, они нигде больше не используются, потому что инструкции `store` ничего не возвращают. Третий аргумент конструктора `StoreInst` определяет изменчивость хранилища – в данном случае в нем передается `false`:

```
StoreInst *st0 = new StoreInst(int32_a, ptrA, false, labelEntry);
st0->setAlignment(4);
StoreInst *st1 = new StoreInst(int32_b, ptrB, false, labelEntry);
st1->setAlignment(4);
```

- Создадим также инструкции `load`, для загрузки значений из стека в `ld0` и `ld1`. Эти значения затем передаются как аргументы инструкции `add`, результат которой, `addRes`, устанавливается как возвращаемое значение функции. Далее функция `makeLLVMModule` вернет модуль LLVM IR с только что созданной функцией `sum`:

```

LoadInst *ld0 = new LoadInst(ptrA, "", false, labelEntry);
ld0->setAlignment(4);
LoadInst *ld1 = new LoadInst(ptrB, "", false, labelEntry);
ld1->setAlignment(4);
BinaryOperator *addRes =
    BinaryOperator::Create(Instruction::Add, ld0, ld1,
        "add", labelEntry);
ReturnInst::Create(mod->getContext(), addRes, labelEntry);
return mod;
}

```

Примечание. Каждая функция создания инструкций имеет множество вариаций. Возможные варианты можно увидеть в заголовочном файле `include/llvm/IR` или в документации *doxygen*.

8. Программа-генератор IR является автономным инструментом, и потому ей нужна функция `main()`. В этой функции мы создадим модуль вызовом `makeLLVMModule` и проверим сконструированный код IR вызовом `verifyModule()`. Элемент `PrintMessageAction` перечисления указывает, что вывод сообщений об ошибках должен производиться в `stderr`. В заключение биткод модуля записывается на диск вызовом функции `WriteBitcodeToFile`, как показано ниже:

```

int main() {
    Module *Mod = makeLLVMModule();
    verifyModule(*Mod, PrintMessageAction);
    std::string ErrorInfo;
    OwningPtr<tool_output_file> Out(new tool_output_file(
        "./sum.bc", ErrorInfo,
        sys::fs::F_None));
    if (!ErrorInfo.empty()) {
        errs() << ErrorInfo << '\n';
        return -1;
    }
    WriteBitcodeToFile(Mod, Out->os());
    Out->keep(); // Объявляет об успехе
    return 0;
}

```

Сборка и запуск генератора IR

Для сборки этого инструмента можно использовать `Makefile` из главы 3, «Инструменты и организация». Наиболее важной частью файла `Makefile` является вызов команды `llvm-config --libs`, которая определяет, с какими библиотеками LLVM будет скомпонован ваш про-

ект. В нашей программе будет использоваться компонент `bitwriter`, вместо компонента `bitreader`, использовавшегося в главе 3, «Инструменты и организация». Поэтому замените команду `llvm-config` на `llvm-config --libs bitwriter core support`. Соберите инструмент, запустите его и проверьте сгенерированный код IR:

```
$ make && ./sum && llvm-dis < sum.bc
...
define i32 @sum(i32 %a, i32 %b) {
entry:
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    %0 = load i32* %a.addr, align 4
    %1 = load i32* %b.addr, align 4
    %add = add i32 %0, %1
    ret i32 %add
}
```

Как генерировать любые конструкции IR с использованием генератора кода C++

Инструмент `llc`, подробно рассматриваемый в главе 6, «Генератор выполняемого кода», имеет одну интересную особенность, помогающую разработчикам генерировать IR. Этот инструмент способен генерировать исходный код на языке C++, эквивалентный заданному файлу LLVM IR (с биткодом в двоичном или текстовом представлении). Это упрощает использование API генератора IR, поскольку позволяет на основе имеющихся файлов IR исследовать приемы конструирования даже самых сложных выражений IR. LLVM реализует данную возможность в виде генератора кода на C++, доступного в инструменте `llc`, если передать ему аргумент `-march=cpp`:

```
$ llc -march=cpp sum.bc -o sum.cpp
```

Откройте получившийся файл `sum.cpp` и обратите внимание, что сгенерированный код на C++ очень похож на тот, что мы написали в предыдущем разделе.

Примечание. Генератор C++ включается по умолчанию, если настроить сборку LLVM с поддержкой всех целей. Однако, если в процессе настройки сборки вы указываете ограниченное множество целей, поддержку генератора C++ нужно включить явно, указав в списке генераторов имя `cpp`: `--enable-targets=x86,arm,mips,cpp`.

Оптимизация на уровне IR

После трансляции в промежуточное представление LLVM IR, программа может подвергаться различным оптимизациям, не зависящим от целевой архитектуры. Оптимизации могут, например, применяться к каждой функции или к каждому модулю по отдельности. В последнем случае применяются межпроцедурные оптимизации. Чтобы усилить воздействие межпроцедурных оптимизаций, пользователь может задействовать инструмент `llvm-link` и скомпоновать несколько модулей LLVM в один. Это позволит расширить область применения оптимизаций; такие оптимизации иногда называют оптимизациями времени компоновки (*link-time optimizations*), потому что они возможны, только в компиляторах, поддерживающих оптимизации, которые простираются за границы единиц трансляции. Все эти оптимизации доступны пользователю LLVM и могут вызываться по отдельности с помощью инструмента `opt`.

Оптимизации времени компиляции и времени компоновки

Инструмент `opt` поддерживает то же множество флагов управления уровнем оптимизации, что и драйвер компилятора Clang: `-O0`, `-O1`, `-O2`, `-O3`, `-Os` и `-Oz`. Clang поддерживает также уровень `-O4`, но `opt` – нет. Флаг `-O4` является синонимом флага `-O3` в комбинации с `-flto` (оптимизация времени компоновки), но, как уже говорилось ранее, поддержка оптимизаций времени компоновки в LLVM зависит от организации входных файлов. Каждый флаг активирует отдельный конвейер оптимизации, включающий определенный комплект оптимизаций, применяемых в определенном порядке. В страницах справочного руководства (*man page*) Clang можно прочитать следующие инструкции:

Флаги -Ox: определяют уровень оптимизации. Флаг -O0 означает «отсутствие оптимизации»: на этом уровне компиляция выполняется быстрее всего и генерируется код, более простой в отладке. Флаг -O2 – уровень умеренной оптимизации, включает большинство доступных оптимизаций. Флаг -Os действует подобно флагу -O2, плюс добавляет дополнительные оптимизации, способствующие уменьшению объема выполняемого кода. Флаг -Oz действует подобно флагу -Os (и, соответственно -O2), но содержит некоторые оптимизации, еще больше уменьшающие объем кода. Флаг

-O3 действует подобно флагу -O2, за исключением того, что включает оптимизации, на выполнение которых требуется больше времени или которые могут приводить к увеличению объема кода (за счет чего обеспечить более высокую скорость выполнения программы). На поддерживаемых платформах флаг -O4 включает оптимизации времени компоновки; объектные файлы сохраняются, преобразуются в формат биткода LLVM и объединяются в один общий файл, после чего выполняются оптимизации времени компоновки. Флаг -O1 занимает промежуточное положение между -O0 и -O2.

Задействовать одну из predetermined последовательностей оптимизаций можно с помощью инструмента `opt`, передав ему файлы с биткодом. Например, следующая команда оптимизирует биткод в файле `sum.bc`:

```
$ opt -O3 sum.bc -o sum-O3.bc
```

Можно также указать флаг, активизирующий стандартные оптимизации времени компиляции:

```
$ opt -std-compile-opts sum.bc -o sum-stdc.bc
```

или задействовать множество стандартных оптимизаций времени компоновки:

```
$ llvm-link file1.bc file2.bc file3.bc -o=all.bc
$ opt -std-link-opts all.bc -o all-stdl.bc
```

Инструмент `opt` поддерживает также возможность выполнения отдельных проходов. Важное место, например, занимает проход `mem2reg`, который преобразует инструкции `alloca` в локальные значения LLVM, с возможным преобразованием их в форму SSA, если они принимают несколько аргументов. В этом случае в преобразование вовлекаются фи-функции (`phi functions`, подробности см. по адресу: http://llvm.org/doxygen/classllvm_1_1PHINode.html) – ими неудобно пользоваться при создании LLVM IR вручную, но они необходимы для поддержки формы SSA. По этой причине предпочтительнее писать не самый оптимальный код, полагающийся на инструкции `alloca`, `load` и `store`, перекладывая создание версии SSA с локальными значениями на проход `mem2reg`. Этот проход отвечал за оптимизацию примера `sum.c` в предыдущем разделе. Например, чтобы выполнить проход `mem2reg` и затем подсчитать число каждой инструкции в модуле, можно выполнить следующую команду (порядок аргументов прохода имеет значение):

```
$ opt sum.bc -mem2reg -instcount -o sum-tmp.bc -stats
```

```
=====
... Statistics Collected ...
=====
```

```
1 instcount - Number of Add insts
1 instcount - Number of Ret insts
1 instcount - Number of basic blocks
2 instcount - Number of instructions (of all types)
1 instcount - Number of non-external functions
2 mem2reg   - Number of alloca's promoted
2 mem2reg   - Number of alloca's promoted with a single store
```

Флаг `-stats` использовался здесь с целью вынудить LLVM вывести статистическую информацию о каждом проходе. В противном случае подсчет инструкций завершится, а на экране ничего не появится.

Добавив флаг `-time-passes` можно также узнать, сколько времени потребовалось на каждую оптимизацию:

```
$ opt sum.bc -time-passes -domtree -instcount -o sum-tmp.bc
```

Полный перечень анализов LLVM, преобразований и дополнительных проходов можно найти по адресу: <http://llvm.org/docs/Passes.html>.

Примечание. Порядок применения оптимизаций к коду оказывает существенное влияние на производительность выполняемого кода и для разных программ порядок применения оптимизаций, дающий наибольший эффект, может отличаться. Используя predetermined последовательности оптимизаций в виде передачи флагов `-Ox`, следует понимать, что они могут оказаться не самыми лучшими для вашей программы. Если вы захотите поэкспериментировать, чтобы почувствовать, насколько сложными могут быть взаимоотношения между оптимизациями, попробуйте вызвать для своего файла инструмент `opt -O3` дважды и посмотреть, как изменится производительность (не обязательно в лучшую сторону) в сравнении с вариантом, когда инструмент `opt -O3` вызывался только один раз.

Определение проходов, имеющих значение

Обычно оптимизации состоят из проходов *анализа* и *преобразования*. На первом определяются параметры и возможности оптимизации, и создаются необходимые структуры данных, которые затем используются на втором проходе. И те и другие реализованы в виде самостоятельных проходов LLVM и могут зависеть от порядка выполнения.

В нашем примере `sum.ll` мы видели, что на уровне `-O0` оптимизации используется несколько инструкций `alloca`, `load` и `store`. Однако, если применить уровень `-O1`, все эти избыточные инструкции исчезнут, потому что уровень `-O1` включает проход `mem2reg`. Но, если бы вы не знали, насколько важен проход `mem2reg`, как бы вы определили, какие проходы имеет смысл применить к вашей программе? Чтобы разобраться в этом вопросе, давайте дадим неоптимизированной версии имя `sum-O0.ll` и оптимизированной – имя `sum-O1.ll`. Чтобы получить оптимизированную версию, соберем ее с флагом `-O1`:

```
$ opt -O1 sum-O0.ll -S -o sum-O1.ll
```

Чтобы получить более полную информацию об изменениях, вызванных каждым преобразованием, можно передать флаг `-print-stats` анализатору `clang` (или флаг `-stats` инструменту `opt`):

```
$ clang -Xclang -print-stats -emit-llvm -O1 sum.c -c -o sum-O1.bc
```

```
=====
... Statistics Collected ...
=====

1 cgsc-passmgr - Maximum CGSCCPassMgr iterations on one SCC
1 functionattrs - Number of functions marked readnone
2 mem2reg      - Number of alloca's promoted with a single store
1 reassociate  - Number of insts reassociated
1 sroa        - Maximum number of partitions per alloca
2 sroa        - Maximum number of uses of a partition
4 sroa        - Number of alloca partition uses rewritten
2 sroa        - Number of alloca partitions formed
2 sroa        - Number of allocas analyzed for replacement
2 sroa        - Number of allocas promoted to SSA values
4 sroa        - Number of instructions deleted
```

Из этого вывода следует, что оба прохода, `mem2reg` и `sroa` (scalar replacement of aggregates – скалярная замена агрегатов), участвовали в удалении избыточных инструкций `alloca`. Чтобы увидеть влияние каждого из проходов в отдельности, выполните только `sroa`:

```
$ opt sum-O0.ll -stats -sroa -o sum-O1.ll
```

```
=====
... Statistics Collected ...
=====

1 cgsc-passmgr - Maximum CGSCCPassMgr iterations on one SCC
1 functionattrs - Number of functions marked readnone
2 mem2reg      - Number of alloca's promoted with a single store
1 reassociate  - Number of insts reassociated
1 sroa        - Maximum number of partitions per alloca
```

2 sroa	- Maximum number of uses of a partition
4 sroa	- Number of alloca partition uses rewritten
2 sroa	- Number of alloca partitions formed
2 sroa	- Number of allocas analyzed for replacement
2 sroa	- Number of allocas promoted to SSA values
4 sroa	- Number of instructions deleted

Обратите внимание, что проход `sroa` использует `mem2reg`, даже при том, что последний не указывался в командной строке явно. Если выполнить только проход `mem2reg`, вы увидите те же самые оптимизации:

```
$ opt sum-00.ll -stats -mem2reg -o sum-01.ll
```

```
=====  
... Statistics Collected ...  
=====
```

```
2 mem2reg - Number of alloca's promoted  
2 mem2reg - Number of alloca's promoted with a single store
```

Зависимости между проходами

Существует два основных вида зависимостей между проходами преобразования и анализа.

- **Явные зависимости:** Проход преобразования запрашивает выполнение анализа и диспетчер проходов автоматически вызывает проход анализа перед преобразованием. Если попытаться выполнить один проход, зависящий от других, диспетчер проходов выполнит все необходимые проходы перед указанным явно. Примерами проходов анализа, предоставляющими информацию для других проходов могут служить **Loop Info** и **Dominator Tree**. Доминаторные деревья (*dominator trees*) – это одна из основных структур данных поддержки алгоритма конструирования SSA и помогает определять, куда помещать фи-функции (*phi functions*). То есть, проход `mem2reg` требует выполнения прохода `domtree`, вследствие чего образуется зависимость между этими двумя проходами:

```
DominatorTree &DT = getAnalysis<DominatorTree>(Func);
```

- **Неявные зависимости:** некоторые проходы преобразования или анализа зависят от использования определенных идиом в коде IR. Благодаря этому легко могут выявляться шаблонные участки, даже при том, что IR имеет мириады других способов выразить одни и те же вычисления. Такая неявная зави-

симость может возникнуть, например, если некоторый проход специально создавался для выполнения после какого-то другого прохода преобразования. То есть, проход может выполнить свою работу, только если код следует определенной идиоме (в результате работы предыдущего прохода). В этом случае, поскольку такого рода зависимости чаще возникают от проходов трансформации, чем от проходов анализа, вам необходимо будет вручную добавлять проходы в очередь в правильном порядке через аргументы командной строки `clang` или `opt`, или использовать *диспетчера проходов*. Если в исходном IR не используются идиомы, определяемые проходом, этот проход просто не выполнит свои преобразования. Стандартные уровни оптимизации уже включают все необходимые оптимизации и не испытывают проблем с зависимостями.

Используя инструмент `opt`, можно получить информацию о том, как диспетчер проходов располагает их, и какие зависимости между проходами удовлетворяет. Например, получить полный список проходов, выполняемых, когда затребован только проход `mem2reg`, можно с помощью следующей команды:

```
$ opt sum-00.ll -debug-pass=Structure -mem2reg -S -o sum-01.ll
```

```
Pass Arguments: -targetlibinfo -datalayout -notti -basictti
-x86tti -domtree -mem2reg -preverify -verify -print-module
```

```
Target Library Information
```

```
Data Layout
```

```
No target information
```

```
Target independent code generator's TTI
```

```
X86 Target Transform Info
```

```
ModulePass Manager
```

```
FunctionPass Manager
```

```
  Dominator Tree Construction
```

```
  Promote Memory to Register
```

```
  Preliminary module verification
```

```
  Module Verifier
```

```
Print module to stderr
```

В списке `Pass Arguments` можно видеть, что диспетчер проходов значительно увеличил число проходов, добавив те, которые необходимы для нормальной работы `mem2reg`. Проход `domtree`, например, требуется проходу `mem2reg`, и потому автоматически был включен в список. Далее, структура вывода подчеркивает иерархическую организацию работы проходов: проходы, перечисленные сразу после

ModulePass Manager, выполняются для каждого модуля в отдельности, тогда как проходы ниже FunctionPass Manager выполняются на уровне отдельных функций. Здесь же можно наблюдать порядок выполнения проходов: проход Promote Memory to Register, например, выполняется после своей зависимости – прохода Dominator Tree Construction.

Прикладной интерфейс проходов

Основой для реализации оптимизаций является класс Pass. Однако он никогда не используется непосредственно – только через его подклассы. Реализуя проход, вы должны выбрать подкласс, лучше соответствующий вашим целям и уровню, на котором ему придется работать, например, на уровне функций, модулей, циклов, связанных компонентов. Ниже перечислены типичные примеры таких подклассов.

- **ModulePass**: один из наиболее универсальных проходов; позволяет проанализировать модуль целиком, вне зависимости от порядка следования функций. Он не гарантирует никаких свойств для своих пользователей, позволяет удалять любые функции и вносить другие изменения. Чтобы воспользоваться им, нужно написать класс, наследующий ModulePass, и переопределить метод `runOnModule()`.
- **FunctionPass**: этот подкласс позволяет обрабатывать функции по отдельности, вне какого-то определенного порядка. Это – наиболее популярный тип проходов. Он запрещает выполнять любые изменения за пределами функций, а также удалять функции и глобальные переменные. Чтобы воспользоваться им, нужно написать наследующий его класс и переопределить метод `runOnFunction()`.
- **BasicBlockPass**: действует на уровне базового блока. Запрещает те же изменения, что и класс FunctionPass, а также изменять или удалять чтобы то ни было за пределами базового блока. Чтобы воспользоваться им, нужно написать класс, наследующий BasicBlockPass, и переопределить метод `runOnBasicBlock()`.

Переопределяемые методы `runOnModule()`, `runOnFunction()` и `runOnBasicBlock()` должны возвращать булево значение `false`, если проанализированная единица (модуль, функция или базовый блок) не изменилась, и `true` – в противном случае. Полное описание

подклассов, наследующих класс `Pass`, можно найти по адресу: <http://llvm.org/docs/WritingAnLLVMPass.html>.

Реализация собственного прохода

Допустим, что нам требуется подсчитать число аргументов в каждой функции, в пределах программы, и вывести числа вместе с именами функций. Напишем для этого проход. Сначала выберем соответствующий подкласс класса `Pass`. Класс `FunctionPass` выглядит наиболее подходящим для нашей цели, так как нам не требуется какой-то определенный порядок анализа функций и не предполагается вносить какие-либо изменения.

Назовем наш проход `FnArgCnt` и поместим соответствующие файлы в дерево с исходными текстами LLVM:

```
$ cd <llvm_source_tree>
$ mkdir lib/Transforms/FnArgCnt
$ cd lib/Transforms/FnArgCnt
```

Создадим файл `FnArgCnt.cpp` в каталоге `lib/Transforms/FnArgCnt` и добавим в него следующую реализацию прохода:

```
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
class FnArgCnt : public FunctionPass {
public:
    static char ID;
    FnArgCnt() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
        errs() << "FnArgCnt --- ";
        errs() << F.getName() << ": ";
        errs() << F.getArgumentList().size() << '\n';
        return false;
    }
};
}

char FnArgCnt::ID = 0;
static RegisterPass<FnArgCnt> X("fnargcnt", "Function Argument
    Count Pass", false, false);
```

Сначала нужно подключить необходимые заголовочные файлы и символы из пространства имен `llvm`:

```
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;
```

Затем объявить класс `FnArgCnt`, наследующий `FunctionPass`, и реализовать основной механизм прохода в методе `runOnFunction()`. Действуя в контексте функции, мы должны вывести имя функции и число принимаемых ею аргументов. Метод всегда возвращает `false`, потому что не вносит никаких изменений. Ниже приводится определение нашего подкласса:

```
namespace {
    struct FnArgCnt : public FunctionPass {
        static char ID;
        FnArgCnt() : FunctionPass(ID) {}

        virtual bool runOnFunction(Function &F) {
            errs() << "FnArgCnt --- ";
            errs() << F.getName() << ": ";
            errs() << F.getArgumentList().size() << '\n';
            return false;
        }
    };
}
```

Значение `ID` определяется внутренними механизмами LLVM для идентификации прохода и ему можно присвоить любое значение:

```
char FnArgCnt::ID = 0;
```

Далее в работу включается механизм регистрации, который регистрирует проход в текущем диспетчере проходов на этапе загрузки прохода:

```
static RegisterPass<FnArgCnt> X("fnargcnt", "Function Argument
    Count Pass", false, false);
```

Первый аргумент, `fnargcnt`, – это имя, используемое инструментом `opt` для идентификации прохода. Второй аргумент содержит расширенное имя. Третий аргумент сообщает, изменяет ли данный проход граф управляющей логики (`Control Flow Graph`, `CFG`) и в последнем передается значение `true`, только если реализуется проход анализа.

Сборка и запуск нового прохода с помощью системы сборки LLVM

Чтобы скомпилировать и установить проход, нужно поместить файл `Makefile` в каталог с исходным кодом. В отличие от предыдущих

проектов, на этот раз мы создали не автономный инструмент, поэтому требуемый файл `Makefile` должен интегрироваться в систему сборки LLVM. Поскольку наш файл `Makefile` теперь опирается на основной файл `Makefile` проекта LLVM, реализующий множество правил, его содержимое выглядит намного проще содержимого `Makefile` для сборки автономного инструмента:

```
# Makefile для прохода FnArgCnt

# Путь к вершине иерархии LLVM

LEVEL = ../../..

# Имя библиотеки для сборки
LIBRARYNAME = LLVMFnArgCnt

# Создаваемая библиотека должна быть превращена в загружаемый
# модуль, чтобы инструменты могли использовать dlopen/dlsym.
LOADABLE_MODULE = 1

# Подключить необходимые определения
include $(LEVEL)/Makefile.common
```

Комментарии в `Makefile` достаточно информативны, и разделяемая библиотека создается с использованием общего для LLVM файла `Makefile`. Благодаря использованию инфраструктуры, наш проход устанавливается вместе с другими стандартными проходами и может непосредственно загружаться инструментом `opt`, но для этого необходимо пересобрать весь проект LLVM.

Нам также хотелось бы, чтобы наш проход компилировался в каталоге с объектными файлами, и нам нужно включить наш проход в файл `Makefile` в каталоге `Transforms`. То есть в файле `lib/Transforms/Makefile` нужно добавить `FnArgCnt` в переменную `PARALLEL_DIRS`:

```
PARALLEL_DIRS = Utils Instrumentation Scalar InstCombine IPO
                Vectorize Hello ObjCARC FnArgCnt
```

В соответствии с инструкциями из главы 1, «Сборка и установка LLVM», проект LLVM требуется повторно сконфигурировать:

```
$ cd каталог-где-будет-выполняться-сборка
$ /ПУТЬ_К_ИСХОДНЫМ_ТЕКСТАМ/configure --prefix=/каталог/для/установки
```

Теперь, находясь в каталоге для объектных файлов, перейдем в каталог нового прохода и выполним `make`:

```
$ cd lib/Transforms/FnArgCnt
$ make
```

Разделяемая библиотека будет помещена в дерево сборки, в каталог `Debug+Asserts/lib`. Здесь `Debug+Asserts` следует заменить именем каталога, соответствующим конфигурации сборки, например, `Release`, если выполнялась сборка в режиме `Release`. Теперь вызовем `opt` с нашим проходом (в `Mac OS X`):

```
$ opt -load <каталог-где-будет-выполняться-сборка>/Debug+Asserts/  
lib/LLVMFnArgCnt.dylib-fnargcnt < sum.bc >/dev/null
```

```
FnArgCnt --- sum: 2
```

При опробовании примера в `Linux` замените расширение в имени файла разделяемой библиотеки на `.so`. Как и ожидалось, наш проход сообщил, что модуль `sum.bc` содержит единственную функцию, которая принимает два целочисленных аргумента, как показано выше.

Точно так же можно пересобрать всю систему LLVM и переустановить ее. Система сборки установит новый выполняемый файл `opt`, который знает о существовании нового прохода и может вызывать его без использования аргумента `-load` командной строки.

Сборка и запуск нового прохода с использованием собственного файла Makefile

Зависимость от системы сборки LLVM кому-то может показаться обременительной, из-за необходимости повторно конфигурировать проект или пересобрать все инструменты LLVM. Если вы относитесь к их числу, вы можете создать автономный `Makefile`, компилирующий проход за пределами дерева с исходными текстами LLVM, как это делалось со всеми нашими проектами прежде. Однако удобство независимости от исходных текстов LLVM иногда стоит дополнительных усилий, которые придется приложить для создания собственного `Makefile`.

В качестве основы для автономного файла `Makefile` мы возьмем файл, использовавшийся для сборки инструментов в главе 3, «Инструменты и организация». Сложность заключается в том, что мы собираемся скомпилировать не инструмент, а разделяемую библиотеку, которая должна загрузиться по требованию инструментом `opt`.

Сначала создадим отдельный каталог для проекта за пределами дерева с исходными текстами LLVM. Поместим в этот каталог файл `FnArgCnt.cpp`. И создадим следующий файл `Makefile`:

```
LLVM_CONFIG?=llvm-config
```

```
ifndef VERBOSE
```

```

QUIET:=@
endif

SRC_DIR?=$(PWD)
LDLFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags)
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)

ifeq ($(shell uname), Darwin)
LOADABLE_MODULE_OPTIONS=-bundle -undefined dynamic_lookup
else
LOADABLE_MODULE_OPTIONS=-shared -Wl,-O1
endif

FNARGPASS=fnarg.so
FNARGPASS_OBJECTS=FnArgCnt.o

default: $(FNARGPASS)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET) $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(FNARGPASS) : $(FNARGPASS_OBJECTS)
    @echo Linking $@
    $(QUIET) $(CXX) -o $@ $(LOADABLE_MODULE_OPTIONS) $(CXXFLAGS)
$(LDLFLAGS) $^

clean::
    $(QUIET) rm -f $(FNARGPASS) $(FNARGPASS_OBJECTS)

```

Новые строки (выделены жирным) в этом файле Makefile, если сравнить его с файлом, взятым из главы 3, «Инструменты и организация» – это условное определение переменной `LOADABLE_MODULE_OPTIONS`, которая содержит параметры командной строки для передачи компоновщику. В ней перечисляются флаги компилятора, зависящие от платформы, которые указывают компилятору, что собирается разделяемая библиотека, а не выполняемый файл. В Linux, например, требуется указать флаг `-shared`, чтобы создать разделяемую библиотеку, а также флаги `-wl, -O1` для передачи флага `-O1` компоновщику GNU `ld`. Эти флаги требуют выполнить оптимизацию таблицы символов, чтобы уменьшить время загрузки библиотеки. Если вы не пользуетесь компоновщиком GNU `ld`, эти флаги можно опустить.

Мы также удалили команду `llvm-config --libs` из командной строки вызова компоновщика. Эта команда использовалась раньше,

чтобы определить список библиотек, с которыми должен компоноваться наш проект. Так как известно, что инструмент `opt` уже хранит все необходимые нам символы, мы просто не подключаем избыточные библиотеки, что ускоряет процесс компоновки.

Сборка нашего проекта теперь может быть выполнена командой:

```
$ make
```

Чтобы задействовать проход, скомпилированный в файл библиотеки `fnarg.so`, достаточно следующей команды:

```
$ opt -load=fnarg.so -fnargcnt < sum.bc > /dev/null
```

```
FnArgCnt --- sum: 2
```

В заключение

Код LLVM IR – это промежуточный пункт между анализатором исходного кода (`frontend`) и генератором выполняемого кода (`backend`). Он находится в позиции, где выполняются оптимизации, независимые от целевой архитектуры. В этой главе мы исследовали инструменты для работы с промежуточным представлением LLVM IR, синтаксис сборок и особенности создания собственных генераторов кода IR. Кроме того, мы узнали, как действует интерфейс проходов, как применяются оптимизации и затем увидели, как писать собственные проходы анализа или преобразования IR.

В следующей главе мы обсудим работу генераторов выполняемого кода (`backends`) LLVM и посмотрим, как можно написать собственный генератор для преобразования кода LLVM IR в выполняемый код для собственной архитектуры.



ГЛАВА 6.

Генератор выполняемого кода

Генератор выполняемого кода (backend) состоит из множества проходов анализа и преобразования, которые превращают код промежуточного представления LLVM IR в объектный (или ассемблерный) код. LLVM поддерживает широкий диапазон целевых архитектур: ARM, AArch64, Hexagon, MSP430, MIPS, Nvidia PTX, PowerPC, R600, SPARC, SystemZ, X86 и XCore. Все эти генераторы имеют общий интерфейс, являющийся частью генератора кода, не зависящего от целевой архитектуры, и прячущий различия архитектур за универсальным API. Чтобы реализовать архитектурно-зависимое поведение, генератор кода должен специализировать обобщенные классы для каждой целевой архитектуры. В этой главе мы охватим множество общих аспектов генераторов выполняемого кода, знание которых пригодится читателям, которые предполагают заняться созданием нового или сопровождением существующего генератора, или разработкой проходов. В частности, мы коснемся следующих тем:

- ♦ обзор организации генераторов выполняемого кода в LLVM;
- ♦ как интерпретировать разные файлы TableGen, описывающие генераторы кода;
- ♦ как происходит выбор инструкций в LLVM;
- ♦ какую роль играют этапы планирования инструкций и распределения регистров;
- ♦ как выполняется эмиссия кода;
- ♦ как написать собственный проход для генератора кода.

Обзор

Преобразование LLVM IR в целевой ассемблерный код выполняется в несколько этапов. Сначала код IR преобразуется в представление

инструкций, функций и глобальных переменных, более дружественное для генератора выполняемого кода. Затем, по мере прохождения разных этапов, это представление постепенно меняется и становится все ближе к инструкциям целевой архитектуры. Диаграмма на рис. 6.1 демонстрирует, какие этапы преобразования должны быть выполнены, чтобы из кода LLVM IR получить объектный или ассемблерный код, а белыми прямоугольниками отмечены точки, где могут выполняться дополнительные проходы оптимизации для дальнейшего улучшения качества трансляции.

Данный конвейер трансляции включает разные этапы создания выполняемого кода, представленные на рис. 6.1 прямоугольниками светло-серого цвета. Иногда их называют *суперпроходами* (*superpasses*), потому что внутренне они реализованы в виде последовательностей более мелких проходов. Разница между светло-серыми и белыми прямоугольниками состоит в том, что первые представляют проходы, успех которых критически важен для успеха всего генератора, тогда как вторые представляют проходы, более важные для увеличения эффективности генерируемого кода. Ниже приводится более подробное описание всех этапов работы генератора кода, изображенных на рис. 6.1.

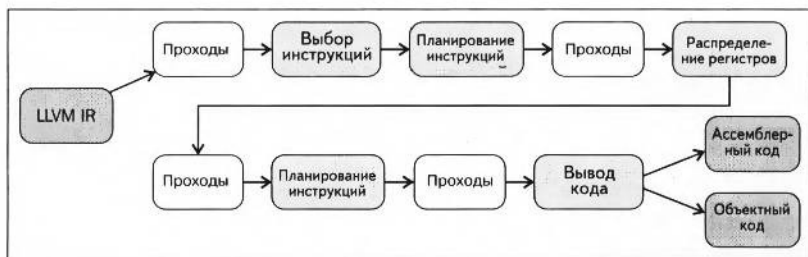


Рис. 6.1. Этапы преобразования кода LLVM IR в объектный код или сборку

На этапе **выбора инструкций (Instruction Selection)** представление IR в памяти преобразуется в узлы SelectionDAG, специфические для заданной архитектуры. Сначала на этом этапе трехадресная структура LLVM IR преобразуется в **ориентированный ациклический граф (Directed Acyclic Graph, DAG)**. Каждый такой граф может представлять вычисления в единственном базовом блоке, соответственно каждый базовый блок ассоциируется с собственным ориентированным ациклическим графом. Узлы графа обычно представляют инструк-

ции, а ребра – потоки информации между ними. Преобразование в граф играет важную роль, так как позволяет библиотеке LLVM генераторов кода использовать алгоритмы выбора инструкций сопоставлением с шаблоном на основе деревьев, с учетом особенностей ориентированных графов. К концу этапа получается граф, в котором все узлы LLVM IR преобразованы в архитектурно-зависимые узлы, то есть в узлы, представляющие машинные инструкции, а не инструкции LLVM.

После этапа выбора инструкций точно известно, какие целевые инструкции будут использоваться для вычислений в каждом базовом блоке, а каждый базовый блок преобразуется в экземпляр класса `SelectionDAG`. Однако нам нужно вернуть трехадресное представление, чтобы определить порядок следования инструкций внутри базовых блоков, потому что ориентированный граф не определяет порядок для инструкций, независимых друг от друга. На первой стадии **планирования инструкций (Instruction Scheduling)**, ее также называют стадией **планирования перед распределением регистров (Pre-register Allocation (RA) Scheduling)**, упорядочивает инструкции, одновременно пытаясь выяснить возможность их параллельного выполнения. Затем инструкции преобразуются в трехадресное представление `MachineInstr`.

Напомним, что LLVM IR имеет бесконечное число регистров. Эта характеристика остается верной вплоть до этапа **распределения регистров (Register Allocation)**, на котором бесконечное множество виртуальных регистров преобразуется в конечное множество регистров целевой архитектуры.

Далее выполняется вторая стадия **планирования инструкций (Instruction Scheduling)**, ее также называют стадией **планирования после распределения регистров (Post-register Allocation (RA) Scheduling)**. Так как на этой стадии доступна информация о фактических регистрах, она может использовать ее для переупорядочения инструкций.

На этапе **эмиссии кода (Code Emission)** инструкции преобразуются из представления `MachineInstr` в экземпляры `MCInst`. После получения этого нового представления, более пригодного для ассемблеров и компоновщиков, появляется две возможности: произвести ассемблерный код или преобразовать двоичные блоки в объектный код определенного формата.

Таким образом, в конвейере генератора выполняемого кода существует четыре разных уровня представления инструкций: пред-

ставление LLVM IR в памяти, узлы SelectionDAG, MachineInstr и MCInst.

Инструменты генераторов кода

Основным инструментом, используемым в качестве генератора выполняемого кода, является `llc`. Продолжая пример с биткодом `sum.bc` из предыдущей главы, соответствующий ему ассемблерный код можно сгенерировать следующей командой:

```
$ llc sum.bc -o sum.s
```

а объектный код – командой:

```
$ llc sum.bc -filetype=obj -o sum.o
```

В этих двух случаях `llc` попытается выбрать генератор выполняемого кода, соответствующий триаде определения платформы в файле `sum.bc`. Чтобы выбрать какой-то другой генератор кода, следует использовать параметр `-march`. Например, следующая команда сгенерирует объектный код для архитектуры MIPS:

```
$ llc -march=mips -filetype=obj sum.bc -o sum.o
```

Если выполнить команду `llc -version`, она выведет полный список поддерживаемых значений параметра `-march`. Обратите внимание, что эти значения совместимы со значениями параметра `--enable-targets`, используемого на этапе конфигурирования LLVM перед сборкой (см. главу 1, «Сборка и установка LLVM»).

Отметьте, однако: здесь мы вынудили `llc` использовать генератор для другой архитектуры, чтобы создать выполняемый код из биткода, первоначально скомпилированного для архитектуры x86. В главе 5, «Промежуточное представление LLVM», мы рассказывали, что промежуточное представление IR обладает некоторыми архитектурно-зависимыми чертами, несмотря на то, что предназначалось на роль универсального языка, исходного для всех генераторов кода. Так как языки C/C++ имеют архитектурно-зависимые атрибуты, эта зависимость не могла не отразиться на LLVM IR.

Поэтому нужно особенно осторожно использовать `llc` с биткодом, в котором триада определения архитектуры не соответствует значению параметра `-march`. В подобных ситуациях могут нарушаться требования ABI, поведение программы может не соответствовать ожидаемому и иногда, генератор оказывается просто не в состоянии сгенерировать выполняемый код. Однако чаще генератор все-таки

производит некоторый код, который, нередко, содержит трудно выявляемые ошибки, что намного хуже.

Примечание. Чтобы увидеть, как зависимость IR от целевой архитектуры может проявляться на практике, рассмотрим небольшой пример. Представьте, что в программе требуется выделить память для вектора указателей на строки и вы использовали типичную для языка C идиому `malloc(sizeof(char*) * n)`. Если вы указали анализатору исходного кода, что целевой, является, например, 32-разрядная архитектура MIPS, он сгенерирует биткод, вызывающий `malloc`, чтобы выделить $n * 4$ байтов памяти, потому что каждый указатель в 32-разрядной архитектуре MIPS занимает 4 байта. Однако, если на основе этого биткода попытаться сгенерировать выполняемый код для архитектуры `x86_64`, вы получите неработающую программу. Во время ее выполнения будут возникать ошибки доступа к памяти, потому что в архитектуре `x86_64` каждый указатель занимает 8 байт, то есть запрограммированный вызов функции `malloc` выделит недостаточный объем памяти, то есть в архитектуре `x86_64` вызов `malloc` должен выделить $n * 8$ байт.

Структура генератора кода

Реализация генератора кода разбросана по нескольким каталогам в дереве с исходными текстами LLVM. Основные библиотеки, используемые генераторами, находятся в каталоге `lib` и в его подкаталогах `CodeGen`, `MC`, `TableGen` и `Target`.

- В каталоге `CodeGen` хранятся заголовочные файлы и файлы с реализацией всех общих алгоритмов создания выполняемого кода: выбора и планирования инструкций, распределения регистров, а также всех необходимых видов анализов.
- В каталоге `MC` хранится реализация низкоуровневых функций для ассемблера и дизассемблера, а также форматов объектных файлов, таких как `ELF`, `COFF`, `MachO` и других.
- В каталоге `TableGen` хранится полная реализация инструмента `TableGen`, используемого для создания исходного кода на языке C++ на основе высокоуровневого описания целевой архитектуры в файле `.td`.
- Поддержка каждой целевой архитектуры хранится в отдельном подкаталоге, в каталоге `Target` (например, `Target/Mips`), в виде нескольких файлов `.cpp`, `.h` и `.td`. Файлы, реализующие схожие функции для разных целей, обычно носят схожие имена.

При создании нового генератора выполняемого кода, вся его реализация помещается только в подкаталог внутри каталога `Target`. Например, рассмотрим организацию генератора для архитектуры `Sparc` в каталоге `Target/Sparc` (см. табл. 6.1).

Таблица 6.1. Организация исходного кода генератора для архитектуры `Sparc`

Имена файлов	Описание
<code>SparcInstrInfo.td</code> <code>SparcInstrFormats.td</code>	Определение инструкций и форматов
<code>SparcRegisterInfo.td</code>	Определения регистров и классов регистров
<code>SparcISelDAGToDAG.cpp</code>	Выбор инструкций
<code>SparcISelLowering.cpp</code>	Упрощение узлов <code>SelectionDAG</code>
<code>SparcTargetMachine.cpp</code>	Информация о свойствах целевой архитектуры, такая как форматы представления данных и ABI
<code>Sparc.td</code>	Определение аппаратных особенностей, разновидностей процессоров и дополнительных возможностей
<code>SparcAsmPrinter.cpp</code>	Реализация эмиссии кода на языке ассемблера
<code>SparcCallingConv.td</code>	Соглашения по вызову функций, соответствующих требованиям ABI

Обычно все генераторы выполняемого кода следуют такой организации, поэтому разработчики легко смогут найти реализацию одной и той же задачи в разных генераторах. Например, если вы описываете регистры для архитектуры `Sparc` в файле `SparcRegisterInfo.td` и вам интересно увидеть, как выглядит такое же описание для архитектуры `x86`, просто загляните в файл `X86RegisterInfo.td` в каталоге `Target/X86`.

Библиотеки генераторов кода

Объем уникального кода `llc` очень невелик (см. `tools/llc/llc.cpp`) и большая часть функциональных возможностей этого инструмента реализована в виде разделяемых библиотек. Этот комплект библиотек делится на две части – зависимых и независимых от целевой архитектуры. Библиотеки, зависимые от целевой архитектуры, хранятся отдельно от независимых, что дает возможность скомпоновать LLVM с ограниченным подмножеством генераторов кода. Например,

если указать параметр `--enable-targets=x86,arm` при настройке конфигурации LLVM, инструмент `llc` будет скомпонован только с библиотеками генераторов для архитектур `x86` и `ARM`.

Напомним, что имена библиотек LLVM начинаются с префикса `libLLVM`. В следующем списке мы опустили этот префикс для большей ясности. Итак, к числу библиотек, независимых от целевой архитектуры, относятся:

- `AsmParser.a`: содержит код для реализации ассемблера и парсинга исходного кода на языке ассемблера;
- `AsmPrinter.a`: содержит функции для реализации вывода кода на языке ассемблера и генератор файлов с исходным кодом на языке ассемблера;
- `CodeGen.a`: содержит реализации алгоритмов для генераторов кода;
- `MC.a`: содержит класс `MCInst` и связанные с ним классы, используемые для представления программ на самом низком уровне;
- `MCDisassembler.a`: содержит код для реализации дизассемблера, который читает объектный код и декодирует байты в объекты `MCInst`;
- `MCJIT.a`: содержит код для реализации динамического (`just-in-time`) генератора кода;
- `MCParser.a`: содержит интерфейс к классу `MCAsmParser` и используется для реализации компонентов, выполняющих парсинг кода на языке ассемблера и часть работы ассемблера;
- `SelectionDAG.a`: содержит класс `SelectionDAG` и связанные с ним классы;
- `Target.a`: содержит интерфейсы, позволяющие алгоритмам, независимым от архитектуры, запрашивать архитектурно-зависимую функциональность, даже при том, что эта функциональность реализована в других библиотеках (архитектурно-зависимых).

К числу библиотек, зависящих от целевой архитектуры, относятся:

- `<Целевая_архитектура>AsmParser.a`: содержит часть библиотеки `AsmParser`, которая зависит от целевой архитектуры, предназначена для реализации ассемблера целевой архитектуры;

- <Целевая_архитектура>AsmPrinter.a: реализует функции для вывода инструкций целевой архитектуры и позволяет генераторам кода выводить файлы с кодом на языке ассемблера;
- <Целевая_архитектура>CodeGen.a: содержит основное множество зависимых функций генератора кода, включая правила обработки регистров, выбор инструкций и планирование инструкций;
- <Целевая_архитектура>Desc.a: содержит информацию о целевой архитектуре с учетом низкоуровневой инфраструктуры MC и реализует регистрацию архитектурно-зависимых объектов MC, таких как MCCodeEmitter;
- <Целевая_архитектура>Disassembler.a: эта библиотека дополняет библиотеку MCDisassembler архитектурно-зависимыми функциями для создания системы, способной читать байты и декодировать их в целевые инструкции MCInst;
- <Целевая_архитектура>Info.a: реализует регистрацию целевой архитектуры в системе LLVM и предоставляет *фасадные* классы, позволяющие архитектурно-независимым библиотекам обращаться к зависимой функциональности.

Префикс <Целевая_архитектура> в именах этих библиотек следует заменить именем целевой архитектуры, например: X86AsmParser.a — это имя библиотеки парсера для архитектуры X86. Все эти библиотеки при установке LLVM помещаются в каталог <ПУТЬ_УСТАНОВКИ_LLVM>/lib.

Язык TableGen

Для описания информации, используемой в LLVM на разных стадиях компиляции, используется специализированный язык TableGen. Например, в главе 4, «Анализатор исходного кода», мы видели примеры использования файлов на языке TableGen (с расширением .td) для описания различных диагностических сообщений. Первоначально язык TableGen был создан разработчиками LLVM с целью помочь программистам в создании своих генераторов кода. Даже при том, что организация библиотек генератора кода ясно подчеркивает отделение целевых характеристик, например, используя разные классы для представления информации о регистрах и об инструкциях, программистам генераторов кода часто приходилось писать код в разных файлах, отражающий одни и те же архитектурные аспекты. Проблема

такого подхода в том, что в коде появляется избыточная информация, которую нужно синхронизировать вручную.

Например, чтобы изменить в генераторе кода алгоритм работы с регистрами, пришлось бы изменить код в нескольких разных местах: в механизме распределения регистров, чтобы показать, какие типы регистров поддерживаются; в механизме вывода ассемблерного кода, чтобы определить имена регистров; в парсере ассемблерного кода, чтобы показать, как выполняется парсинг имен регистров; и в дисассемблере, который должен знать, как кодировать регистры. Это существенно усложняет поддержку генераторов кода.

Чтобы улучшить ситуацию, был создан декларативный язык TableGen, предназначенный для описания файлов, играющих роль централизованного репозитория информации о целевой архитектуре. Идея состояла в том, чтобы сосредоточить описание некоторых аспектов аппаратной архитектуры в одном месте, например, описание машинных инструкций в файле `<Целевая_архитектура>InstrInfo.td`, и затем использовать этот репозиторий в генераторе кода, например, в реализации алгоритма выбора инструкций, слишком длинной, чтобы писать ее вручную.

В настоящее время язык TableGen применяется для описания всех видов информации о целевой архитектуре, таких как форматы инструкций, инструкции, регистры, ориентированные ациклические графы (Directed Acyclic Graph, DAG) для сопоставления, порядок выбора инструкций, соглашения о вызовах и свойства процессоров (поддерживаемая структура системы команд (Instruction Set Architecture, ISA) и семейства процессоров).

Примечание. Полное и автоматическое создание генераторов кода, симуляторов и файлов с описанием процессоров было долгожданной целью в научных кругах, занимающихся исследованием компьютерных архитектур, и эта проблема все еще остается открытой. Типичный подход к ее решению заключается в описании всей информации об архитектуре на декларативном языке, похожем на TableGen, и использовании инструментов, которые пытаются сгенерировать все виды программного обеспечения, необходимого для оценки и тестирования процессорной архитектуры. Как можно догадаться, данная проблема не имеет простого решения и качество инструментов, сгенерированных автоматически, значительно уступает качеству инструментов, написанных вручную. Подход на основе применения TableGen был предпринят в LLVM с целью освободить программиста от некоторых рутинных задач, но оставить за ним полный контроль над реализацией нестандартной логики на языке C++.

Язык

Язык TableGen состоит из определений и классов, используемых для формирования записей. Определение `def` используется для создания записей из ключевых слов `class` и `multiclass`. Далее эти записи обрабатываются инструментами TableGen, и на их основе создаются: информация для генераторов кода, диагностические сообщения для Clang, параметры драйвера Clang и функции проверки для статического анализатора. То есть, фактическое значение этим записям придают конкретные генераторы кода, а сами записи хранят лишь информацию.

Давайте рассмотрим простой пример, иллюстрирующий, как действует TableGen. Допустим, что нам требуется определить инструкции `ADD` и `SUB` для гипотетической архитектуры, где инструкция `ADD` имеет две формы: одна форма, когда все операнды являются регистрами, и другая, когда один операнд – регистр, а другой – непосредственное значение.

Инструкция `SUB` имеет только одну форму – первую. Взгляните на следующий пример кода из файла `insns.td`:

```
class Insn<bits <4> MajOpc, bit MinOpc> {
  bits<32> insnEncoding;
  let insnEncoding{15-12} = MajOpc;
  let insnEncoding{11} = MinOpc;
}
multiclass RegAndImmInsn<bits <4> opcode> {
  def rr : Insn<opcode, 0>;
  def ri : Insn<opcode, 1>;
}
def SUB : Insn<0x00, 0>;
defm ADD : RegAndImmInsn<0x01>;
```

Класс `Insn` представляет обычную инструкцию, а мультикласс `RegAndImmInsn` – инструкции, имеющие формы, упомянутые выше. Конструкция `def SUB` определяет запись `SUB`, а конструкция `defm ADD` – две записи: `ADDRr` и `ADDRi`. С помощью инструмента `llvm-tblgen` можно обработать файл `.td` и проверить получившиеся записи:

```
$ llvm-tblgen -print-records insns.td
----- Classes -----
class Insn<bits<4> Insn:MajOpc = { ?, ?, ?, ? }, bit Insn:MinOpc = ?> {
  bits<5> insnEncoding = { Insn:MinOpc, Insn:MajOpc{0},
  Insn:MajOpc{1}, Insn:MajOpc{2}, Insn:MajOpc{3} } ;
  string NAME = ?;
}
----- Defs -----
```



```
def ADDri { // Insn ri
  bits<5> insnEncoding = { 1, 1, 0, 0, 0 };
  string NAME = "ADD";
}

def ADDrr { // Insn rr
  bits<5> insnEncoding = { 0, 1, 0, 0, 0 };
  string NAME = "ADD";
}

def SUB { // Insn
  bits<5> insnEncoding = { 0, 0, 0, 0, 0 };
  string NAME = ?;
}
```

Инструмент `llvm-tblgen` позволяет также использовать генераторы кода TableGen. Если выполнить команду `llvm-tblgen --help`, она выведет список всех доступных генераторов. Обратите внимание, что пример выше не использует никаких генераторов кода. Дополнительную информацию о языке TableGen можно найти на странице <http://llvm.org/docs/TableGenFundamentals.html>.

Использование файлов `.td` с генераторами кода

Как отмечалось выше, генераторы кода широко используют записи на языке TableGen, содержащие информацию о целевой архитектуре. В этом подразделе мы покажем несколько примеров использования файлов TableGen для целей создания выполняемого кода.

Свойства целевой архитектуры

Файлы с именами вида `<Целевая_архитектура>.td` (например, `x86.td`) определяют поддерживаемую структуру системы команд (Instruction Set Architecture, ISA) и семейства процессоров. Например, `x86.td` определяет расширение AVX2:

```
def FeatureAVX2 : SubtargetFeature<"avx2", "x86SSELevel", "AVX2",
  "Enable AVX2 instructions",
  [FeatureAVX]>;
```

Ключевое слово `def` определяет запись `FeatureAVX2` на основе класса записей `SubtargetFeature`. Последний аргумент – это список других особенностей, уже включенных в определение. То есть, процессор с расширением AVX2 поддерживает все инструкции AVX.

Кроме того, можно также определить тип процессора и указать, какие расширения ISA или особенности он поддерживает:

```
def : ProcessorModel<"corei7-avx", SandyBridgeModel,  
    [FeatureAVX, FeatureCMPXCHG16B, ...,  
    FeaturePCLMUL]>;
```

Файлы <Целевая_архитектура>.td подключают также все другие файлы .td и являются главными источниками информации для своих архитектур. Инструменту llvm-tblgen всегда следует передавать такие файлы, чтобы он мог извлекать требуемые записи TableGen. Например, вывести все записи для архитектуры x86 можно следующими командами:

```
$ cd <llvm_source>/lib/Target/X86  
$ llvm-tblgen -print-records X86.td -I ../../../../include
```

В файле x86.td содержится часть информации, которую TableGen использует для создания файла x86GenSubtargetInfo.inc, но не ограничивается им и, вообще говоря, нет прямой связи между единственным файлом .td и единственным файлом .inc. Проще говоря, файл <Целевая_архитектура>.td является файлом верхнего уровня, который подключает все другие с помощью директив include языка TableGen. Соответственно, когда генерируется код на C++, TableGen всегда анализирует все файлы .td, позволяя вам помещать свои записи туда, куда сочтете нужным. Несмотря на то, что x86.td подключает все остальные файлы .td, содержимое этого файла, помимо директив include, соответствует определению подкласса Subtarget для архитектуры x86.

Если заглянуть в файл x86Subtarget.cpp, реализующий класс x86Subtarget, можно увидеть директиву препроцессора C++ #include "X86GenSubtargetInfo.inc", из чего становится понятно, как выполняется встраивание кода на C++, сгенерированного TableGen, в обычный исходный код. В частности, данный подключаемый файл содержит константы, определяющие свойства процессора, вектор свойств, связанный со строкой описания и другие ресурсы.

Регистры

Регистры и классы регистров определяются в файле <Целевая_архитектура>RegisterInfo.td. Классы регистров используются далее в определениях инструкций для связывания операндов инструкций с конкретным набором регистров. Например, 16-разрядные регистры определены в файле x86RegisterInfo.td с применением следующей идиомы:

```
let SubRegIndices = [sub_8bit, sub_8bit_hi], ... in {
  def AX : X86Reg<"ax", 0, [AL,AH]>;
  def DX : X86Reg<"dx", 2, [DL,DH]>;
  def CX : X86Reg<"cx", 1, [CL,CH]>;
  def BX : X86Reg<"bx", 3, [BL,BH]>;
  ...
}
```

Конструкция `let` используется для определения дополнительного поля, в данном случае `SubRegIndices`, которое включается во все записи, находящиеся внутри фигурных скобок `{ }`. Определения 16-разрядных регистров являются производными от класса `X86Reg` и хранят имя регистра, число и список 8-разрядных подрегистров. Определение класса 16-разрядных регистров приводится ниже:

```
def GR16 : RegisterClass<"X86", [i16], 16,
  (add AX, CX, DX, ..., BX, BP, SP,
   R8W, R9W, ..., R15W, R12W, R13W)>;
```

Класс регистров `GR16` содержит все 16-разрядные регистры и соответствующий им предпочтительный порядок распределения. После обработки кода `TableGen` в имя каждого класса регистров добавляется окончание `RegClass`, например, имя `GR16` превращается в `GR16RegClass`. `TableGen` генерирует определения регистров и классов регистров, реализации методов для извлечения информации о них, двоичное представление для ассемблера и информацию DWARF (Debug With Arbitrary Record Format – информация для отладки с записями в свободном формате). Проверить код, сгенерированный на основе определений `TableGen` можно с помощью `llvm-tblgen`:

```
$ cd <llvm_source>/lib/Target/X86
$ llvm-tblgen -gen-register-info X86.td -I ../../../../include
```

Точно так же можно проверить файл `<КАТАЛОГ_СБОРКИ_LLVM>/lib/Target/X86/X86GenRegisterInfo.inc` с исходным кодом на C++, сгенерированным в процессе сборки LLVM. Этот файл подключается файлом `X86RegisterInfo.cpp`, чтобы упростить определение класса `X86RegisterInfo`. Кроме всего прочего он содержит перечисление с регистрами процессора, что делает данный файл удобным справочником во время отладки генератора, помогающим вспомнить, как называется регистр, представленный, например, номером 16.

Инструкции

Форматы инструкций определяются в файле `<Целевая_архитектура>InstrFormats.td`, а сами инструкции – в файле `<Целевая_архитектура>InstrInfo.td`. Форматы инструкций опре-

деляют поля, необходимые для представления инструкций в двоичном формате, а записи с определениями инструкций представляют каждую инструкцию в отдельности. Вы можете создать промежуточные классы инструкций, то есть, классы TableGen, используемые в определениях записей инструкций, чтобы вынести за скобки общие свойства, например, для кодирования похожих инструкций обработки данных. Однако каждая инструкция или формат должна прямо или косвенно наследовать класс TableGen Instruction, определение которого находится в файле include/llvm/Target/Target.td. Поля этого класса наглядно показывают, какие поля в записях инструкций ожидают получить генераторы кода TableGen:

```
class Instruction {
  dag OutOperandList;
  dag InOperandList;
  string AsmString = "";
  list<dag> Pattern;
  list<Register> Uses = [];
  list<Register> Defs = [];
  list<Predicate> Predicates = [];
  bit isReturn = 0;
  bit isBranch = 0;
  ...
}
```

dag – специальный тип TableGen, используемый для хранения узлов SelectionDAG. Эти узлы представляют коды операций, регистры или константы на этапе выбора инструкции. Поля в примере выше играют следующие роли:

- Поле OutOperandList хранит получившиеся на выходе узлы, давая возможность генератору кода идентифицировать узлы DAG, представляющие инструкцию. Например, для инструкции ADD в архитектуре MIPS это поле определяется как (outs GPR32Opnd:\$rd), где:
 - outs – специальный узел DAG, обозначающий, что его потомками являются операнды;
 - GPR32Opnd – узел DAG, характерный для MIPS, обозначающий экземпляр 32-разрядного регистра общего назначения в архитектуре MIPS;
 - \$rd – произвольное имя регистра, используемое для идентификации узла.
- Поле InOperandList хранит входные узлы. Например, для инструкции ADD в архитектуре MIPS это поле определяется как "(ins GPR32Opnd:\$rs, GPR32Opnd:\$rt)".

- Поле `AsmString` представляет исходный код инструкции на языке ассемблера, например, для инструкции `ADD` в архитектуре MIPS это поле получит значение `"add $rd, $rs, $rt"`.
- Поле `Pattern` – список объектов `dag`, которые будут использоваться для сопоставления в ходе выбора инструкций. При совпадении с шаблоном, механизм выбора инструкций замещает совпавшие узлы этой инструкцией. Например, в шаблоне `[(set GPR32Opnd:$rd, (add GPR32Opnd:$rs, GPR32Opns:$rt))]` инструкции `ADD` архитектуры MIPS, квадратные скобки – [и] – отмечают содержимое списка, имеющего единственный элемент `dag`, который определяется внутри круглых скобках в LISP-подобной нотации.
- Поля `Uses` и `Defs` – списки регистров, неявно используемых (`Uses`) и определяемых (`Defs`) в ходе выполнения этой инструкции. Например, инструкция возврата из подпрограммы в архитектуре RISC неявно использует регистр адреса возврата, а инструкция вызова подпрограммы – неявно определяет регистр адреса возврата.
- Поле `Predicates` хранит список предварительных условий, которые должны быть проверены перед попыткой сопоставления с шаблоном на этапе выбора инструкций. Если проверка потерпит неудачу, сопоставление не выполняется. Например, условие может указывать, что инструкция действительна только для определенной версии архитектуры. Если в триаде определения архитектуры указана другая версия, этот предикат вернет ложное значение и инструкция не будет выбрана.
- Другие поля, включая `isReturn` и `isBranch`, несут дополнительную информацию о поведении инструкции. Например, если `isBranch = 1`, генератор кода будет знать, что инструкция является инструкцией ветвления и должна завершать базовый блок.

В следующем фрагменте демонстрируется определение инструкции `XNORrr` в файле `SparcInstrInfo.td`. Определение имеет формат `F3_1` (определяется в `SparcInstrFormats.td`), который охватывает часть формата `F3`, описанного в руководстве по архитектуре SPARC V8:

```
def XNORrr : F3_1<2, 0b000111,
  (outs IntRegs:$dst), (ins IntRegs:$b, IntRegs:$c),
  "xnor $b, $c, $dst",
```

```
[(set i32:$dst, (not (xor i32:$b, i32:$c)))]>;
```

Инструкция `XNORrr` имеет два операнда `IntRegs` (узел DAG, представляющий класс 32-разрядных целочисленных регистров в архитектуре SPARC) и один результат `IntRegs`, как, например, в: `OutOperandList = (outs IntRegs:$dst)` и `InOperandList = (ins IntRegs:$b, IntRegs:$c)`.

Ассемблерный код `AsmString` ссылается на указанные операнды с использованием префикса `$`: `"xnor $b, $c, $dst"`. Список `Pattern` элементов `(set i32:$dst, (not (xor i32:$b, i32:$c)))` содержит узлы `SelectionDAG`, которые должны соответствовать инструкции. Например, инструкция `XNORrr` соответствует любому выражению, где результат `xor` инвертируется с помощью `not` и оба операнда являются регистрами.

Проверить поля записи для инструкции `XNORrr` можно с помощью следующей последовательности команд:

```
$ cd <llvm_source>/lib/Target/Sparc
$ llvm-tblgen -print-records Sparc.td -I ../../../../include |
grep XNORrr -A 10
```

Многие генераторы кода TableGen используют информацию из записей инструкций в своей работе, генерируя разные файлы `.inc` на основе одни и тех же записей. Это делается с целью создания централизованного репозитория, используемого разными частями генератора кода. Ниже перечислены файлы, сгенерированные разными генераторами TableGen:

- `<Целевая_архитектура>GenDAGISel.inc`: использует информацию из поля `patterns` записей инструкций для получения кода, отбирающего инструкции из структуры `SelectionDAG`. Этот файл подключается в `<Целевая_архитектура>ISelDAGtoDAG.cpp`.
- `<Целевая_архитектура>GenInstrInfo.inc`: содержит перечисление со списком всех инструкций для заданной целевой архитектуры, а также ряд таблиц с описаниями инструкций. Подключается в файлах `<Целевая_архитектура>InstrInfo.cpp`, `<Целевая_архитектура>InstrInfo.h`, `<Целевая_архитектура>MCTargetDesc.cpp` и `<Целевая_архитектура>MCTargetDesc.h`. Но, перед подключением файла, созданного TableGen, каждый из перечисленных файлов определяет собственный набор макросов, влияющих на парсинг подключаемых файлов.

- <Целевая_архитектура>GenAsmWriter.inc: содержит код, отображающий инструкции в строки с кодом на языке ассемблера. Подключается в <Целевая_архитектура>AsmPrinter.cpp.
- <Целевая_архитектура>GenCodeEmitter.inc: содержит код, отображающий инструкции в двоичный код, то есть генерирует машинный код для заполнения объектного файла. Подключается в <Целевая_архитектура>CodeEmitter.cpp.
- <Целевая_архитектура>GenDisassemblerTables.inc: реализует таблицы и алгоритмы декодирования последовательности байтов и идентификации представляемых ими инструкций. Используется для реализации дизассемблеров и подключается в <Целевая_архитектура>Disassembler.cpp.
- <Целевая_архитектура>GenAsmMatcher.inc: реализует парсинг ассемблерного кода. Дважды подключается в <Целевая_архитектура>AsmParser.cpp, каждый раз с разными наборами макросов препроцессора, влияющими на парсинг этого файла.

Этап выбора инструкций

Выбор инструкций – это процесс преобразования LLVM IR в узлы SelectionDAG (SDNode), представляющие целевые инструкции. Сначала на основе инструкций LLVM IR конструируется граф DAG, создается объект SelectionDAG, узлы которого отображаются в операции IR. Затем эти узлы проходят через этапы упрощения, объединения DAG и легализации, чтобы упростить сопоставление с целевыми инструкциями. Затем выполняется преобразование графа DAG с использованием сопоставления с шаблоном, в ходе которого узлы SelectionDAG преобразуются в узлы, представляющие целевые инструкции.

Примечание. Проход выбора инструкций является одним из самых ресурсоемких. Измерение производительности компиляции тестовых функций из SPEC CPU2006 показывает, что в среднем на долю одного только прохода выбора инструкций приходится почти половина времени выполнения инструмента `llc` с флагом `-O2`, генерирующего код для архитектуры `x86` в LLVM 3.0. Если вам интересно узнать среднее время, затрачиваемое всеми архитектурно-зависимыми и архитектурно-независимыми проходами с флагом `-O2`, обращайтесь к приложению отчета с результатами тестирования JIT-компилятора LLVM: <http://www.ic.unicamp.br/~reltech/2013/13-13.pdf>.

Класс SelectionDAG

Класс SelectionDAG использует DAG для представления вычислений в каждом базовом блоке, и каждый узел SDNode соответствует инструкции или операнду. На рис. 6.2 показана диаграмма, сгенерированная LLVM и изображающая граф DAG для файла sum.bc, в котором имеется только одна функция с единственным базовым блоком.

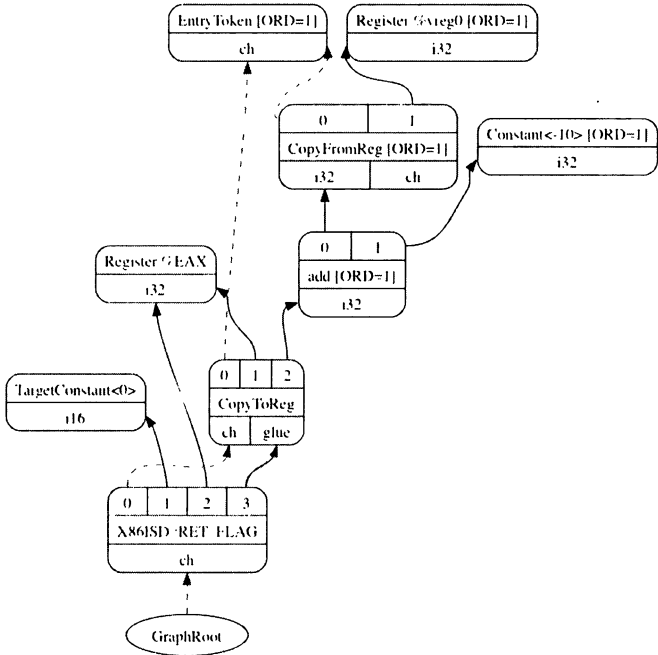


Рис. 6.2. Граф DAG для файла sum.bc с единственной функцией и единственным базовым блоком

Ребра в этом графе DAG определяют порядок операций посредством отношений использование-определение. Если узел В (например, add) имеет исходящее ребро, связывающее его с узлом А (например, Constant<-10>), это означает, что узел А определяет значение (32-разрядное целое со знаком -10), которое используется узлом В (как операнд в операции сложения). То есть, операция А должна выполняться перед операцией В. Черные стрелки представляют обычные ребра, отражающие потоки данных. Пунктирные синие стрелки

представляют *цепочки*, не связанные с данными, которые определяют порядок выполнения не связанных между собой инструкций. Например, инструкции `load` и `store` должны выполняться в том порядке, в каком они указаны в исходной программе, если обращаются к одним и тем же ячейкам в памяти. На рис. 6.2 видно, что операция `CopyToReg` должна выполняться перед операцией `X86ISD::RET_FLAG`, так как они связаны синей стрелкой. Красная стрелка *связывает* (*glue*) смежные узлы, в том смысле, что такие узлы должны следовать друг за другом, без промежуточных инструкций между ними. Например, мы видим, что те же самые узлы `CopyToReg` и `X86ISD::RET_FLAG` должны планироваться для выполнения непосредственно друг за другом, потому что они связаны красной стрелкой.

Узлы-поставщики могут предоставлять значения разных типов, в зависимости от отношений с узлами-потребителями. Значение необязательно должно быть конкретным – значением может быть и абстрактная лексема – и может быть значением одного из следующих типов.

- Тип конкретного значения: целочисленный, вещественный, вектор или указатель. Примером значения такого типа может служить результат, возвращаемый узлом обработки данных, который вычисляет новое значение на основании своих операндов. Тип может быть: `i32`, `i64`, `f32`, `v2f32` (вектор с двумя элементами типа `f32`), `iPTR` и другие. Когда значение потребляется другим узлом, отношение производитель-потребитель обозначается на диаграмме LLVM обычным ребром черного цвета.
- Тип `Other`. Это абстрактная лексема, представляющая значение-цепочку (*ch* на рис. 6.2). Когда другой узел потребляет значение типа `Other`, отношение между узлами обозначается на диаграмме LLVM ребром в виде пунктирной стрелки синего цвета.
- Тип `Glue` представляет связи. Когда другой узел потребляет значение типа `Glue`, связь между узлами обозначается на диаграмме LLVM ребром в виде стрелки красного цвета.

Объекты `SelectionDAG` имеют специальный узел `EntryToken`, отмечаящий точку входа в базовый блок, который поставляет значение типа `Other`, чтобы позволить узлам в цепочке получить эту первую лексему. Объект `SelectionDAG` имеет также ссылку на корень графа, следующий сразу за последней инструкцией, который также интерпретируется как значение-цепочка типа `Other`.

На этой стадии архитектурно-зависимые и архитектурно-независимые узлы могут сосуществовать вместе, как результат предшествующих шагов, таких как упрощение и легализация, которые отвечают за подготовку DAG к выбору инструкций. Однако после выбора инструкций все узлы, соответствующие целевым инструкциям, превратятся в архитектурно-зависимые. В диаграмме рис. 6.2 имеются следующие архитектурно-независимые узлы: `CopyToReg`, `CopyFromReg`, `Register(%vreg0)`, `add` и `Constant`. Помимо них имеются также предварительно обработанные и архитектурно-зависимые узлы (которые, впрочем, могут измениться после выбора инструкций): `TargetConstant`, `Register(%EAX)` и `X86ISD::RET_FLAG`.

На диаграмме можно также наблюдать следующую семантику:

- `Register`: этот узел может ссылаться на виртуальный или физический (архитектурно-зависимый) регистр;
- `CopyFromReg`: этот узел копирует регистр, определенный за пределами текущего базового блока, позволяя использовать его в текущем контексте – в нашем примере он копирует аргумент функции;
- `CopyToReg`: этот узел копирует значение в указанный регистр, не возвращая никакого конкретного значения для других узлов. Однако он производит значение-цепочку (типа `Other`) для включения в цепочку операций, наряду с другими узлами, так же не генерирующими конкретного значения. Например, чтобы использовать значение, записанное в `EAX`, узел `X86ISD::RET_FLAG` использует результат типа `i32`, поставляемый узлом `Register(%EAX)`, и потребляет значение-цепочку, произведенное узлом `CopyToReg`. Тем самым гарантируется, что значение `%EAX` будет обновлено операцией `CopyToReg`, потому что порядок следования в цепочке требует, чтобы инструкция `CopyToReg` выполнялась перед инструкцией `X86ISD::RET_FLAG`.

Дополнительные детали, касающиеся класса `SelectionDAG`, ищите в заголовочном файле `llvm/include/llvm/CodeGen/SelectionDAG.h`. Типы результатов, производимых узлами, можно найти в заголовочном файле `llvm/include/llvm/CodeGen/ValueTypes.h`. Определения архитектурно-независимых узлов находятся в заголовочном файле `llvm/include/llvm/CodeGen/ISDOpcodes.h`, а определения архитектурно-независимых узлов – в заголовочном файле `lib/Target/<Целевая_архитектура>/<Целевая_архитектура>ISelLowering.h`.

Упрощение

В предыдущем подразделе была показана диаграмма, в которой одновременно существуют архитектурно-зависимые и архитектурно-независимые узлы. Возникает закономерный вопрос: «Как некоторые архитектурно-зависимые узлы оказались в классе `SelectionDAG`, если он служит исходной информацией для этапа выбора инструкций?». Прежде чем разбираться с этим, взгляните на рис. 6.3 с диаграммой, где изображены все этапы, предшествующие выбору инструкций, начиная с этапа создания LLVM IR.

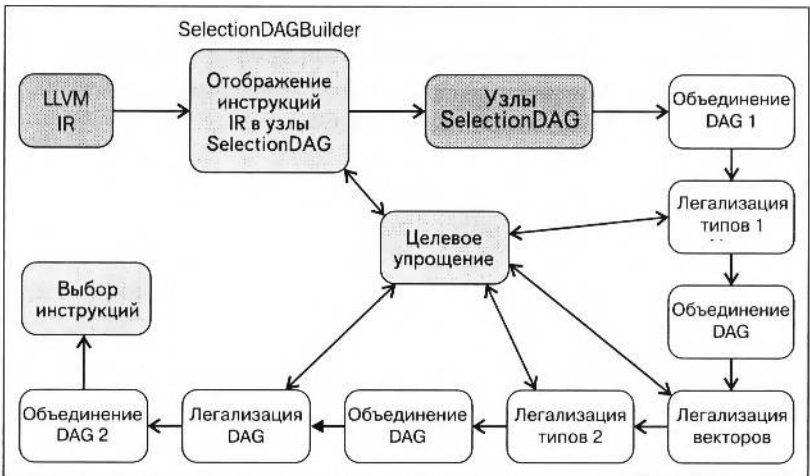


Рис. 6.3. Этапы, предшествующие выбору инструкций

Во-первых, экземпляр `SelectionDAGBuilder` (см. `SelectionDAG-ISel.cpp`) обходит все функции и создает объекты `SelectionDAG` для каждого базового блока. В ходе этого процесса некоторые специальные инструкции IR, такие как `call` и `ret`, уже требуют применения архитектурно-зависимых идиом – например, передача аргументов в вызов функции и возврат результатов – для преобразования в узлы `SelectionDAG`. Чтобы решить данную проблему, на этом этапе впервые используются алгоритмы из класса `TargetLowering`. Этот класс является абстрактным интерфейсом, который должен быть реализован для каждой целевой архитектуры, помимо массы других, универсальных функций, используемых всеми генераторами кода.

С целью реализации этого абстрактного интерфейса для каждой целевой архитектуры объявляется подкласс с именем <Целе-

вая_архитектура>TargetLowering, наследующий TargetLowering. В каждом таком подклассе переопределяются методы, реализующие упрощение конкретных, архитектурно-независимых *высокоуровневых* узлов до уровня, близкого к машинному. Как можно догадаться, лишь малая часть узлов требует такого упрощения, а подавляющая масса узлов преобразуется на этапе выбора инструкций. Например, в объекте SelectionDAG, полученном из sum.bc, для упрощения IR-инструкции ret используется метод X86TargetLowering::LowerReturn() (см. lib/Target/X86/X86ISelLowering.cpp). В результате этого получается узел X86ISD::RET_FLAG, который копирует результат функции в EAX – архитектурно-зависимый способ возврата результата из функции.

Объединение DAG и легализация

Объект SelectionDAG, сконструированный экземпляром SelectionDAGBuilder, еще не готов к выбору инструкций и должен пройти дополнительные преобразования – они показаны на рис. 6.3. Эти преобразования выполняются в следующем порядке:

- Проход объединения DAG оптимизирует недостаточно оптимальные конструкции SelectionDAG, путем сопоставления множеств узлов и заменой их более простыми конструкциями. Например, фрагмент графа (add (Register X), (constant 0)) можно *свернуть* до (Register X). Аналогично архитектурно-зависимые методы объединения могут выявлять группы узлов и определять необходимость свертки и объединения инструкций для данной архитектуры. Обобщенная реализация объединения DAG находится в файле lib/CodeGen/SelectionDAG/DAGCombiner.cpp, а архитектурно-зависимая – в файле lib/Target/<Целевая_архитектура>/<Целевая_архитектура>ISelLowering.cpp. Метод setTargetDAGCombine() отмечает узлы, которые могут быть объединены. В архитектуре MIPS, к примеру, выполняется попытка объединить операции сложения – см. setTargetDAGCombine(ISD::ADD) и performADDCombine() в lib/Target/Mips/MipsISelLowering.cpp.

Примечание. Этап объединения DAG выполняется после каждого этапа легализации для устранения избыточности в SelectionDAG. Кроме того, этому этапу доступна информация о его месте в цепочке проходов (например, после легализации типов или после лега-

лизации векторов) и эта информация может использоваться для выбора оптимальных решений.

- Проход легализации типов гарантирует, что этапу выбора инструкций придется иметь дело только с *легальными* типами. Легальными называются типы, поддерживаемые целевой архитектурой. Например, операция сложения операндов типа `i64` считается *нелегальной* в архитектурах, поддерживающих только тип `i32`. В этом случае этап легализации типов разобьет операнд `i64` на два операнда `i32` и сгенерирует соответствующие узлы для их обработки. Целевые архитектуры определяют классы регистров, связанные с каждым типом, явно объявляя поддерживаемые типы. То есть, нелегальные типы должны быть выявлены и обработаны соответствующим образом – см. определения в файле `llvm/include/llvm/Target/TargetLowering.h`. И снова, для целевых архитектур могут быть реализованы специализированные методы легализации типов. Процедура легализации типов выполняется дважды: после первого объединения DAG и после легализации векторов.
- Иногда бывает так, что генератор кода напрямую поддерживает векторы, в том смысле, что определяет специальный класс регистров для этого, а конкретная операция с заданным типом вектора – нет. Например, архитектура `x86` с расширением `SSE2` поддерживает векторный тип `v4i32`. Однако в ней отсутствует инструкция для поддержки операции `ISD::OR` с типами `v4i32` – только с типами `v2i64`. Процедура легализации векторов обрабатывает подобные ситуации и преобразует инструкции под поддерживаемые типы. Кроме того, некоторые архитектуры могут предусматривать собственные варианты легализации. В случае с упомянутой операцией `ISD::OR` выполняется ее расширение для использования типа `v2i64`. Взгляните на следующий фрагмент в файле `lib/Target/X86/X86ISelLowering.cpp`:

```
setOperationAction(ISD::OR, v4i32, Promote);
AddPromotedToType (ISD::OR, v4i32, MVT::v2i64);
```

Примечание. В некоторых случаях расширение заключается в устранении векторов и их замене скалярными значениями, что в свою очередь может привести к неподдерживаемым скалярным типам для данной архитектуры. Однако последующая операция легализации типов устранил это несоответствие.

- Легализация DAG играет ту же роль, что и легализация векторов, но обрабатывает все остальные операции с неподдерживаемыми типами (скалярными или векторными). Она точно также выполняет замену типов и обработку нестандартных узлов. Например, архитектура x86 не поддерживает ни одну из следующих трех инструкций: инструкция преобразования целого со знаком в вещественное (`ISD::SINT_TO_FP`) для типа `i8`, которая замещается инструкцией с более широким типом; инструкция целочисленного деления со знаком (`ISD::SDIV`) для операндов `i32`, которая замещается *вызовом библиотечной функции* деления; и инструкция получения абсолютного значения вещественного числа (`ISD::FABS`), которая замещается группой инструкций, дающих тот же эффект. В архитектуре x86 такие действия (см. `lib/Target/X86/X86ISelLowering.cpp`) реализованы, как показано ниже:

```
setOperationAction(ISD::SINT_TO_FP, MVT::i8, Promote);
setOperationAction(ISD::SDIV, MVT::i32, Expand);
setOperationAction(ISD::FABS, MVT::f32, Custom);
```

Выбор инструкций с преобразованием «DAG-to-DAG»

Цель этапа выбора инструкций с преобразованием «DAG-to-DAG» заключается в трансформации архитектурно-независимых узлов в архитектурно-зависимые. Алгоритм выбора инструкций является локальным, в том смысле, что обрабатывает экземпляры `SelectionDAG` (базовые блоки) по одному.

Для примера ниже (рис. 6.4) изображена окончательная структура `SelectionDAG`, получившаяся после этапа выбора инструкций. Узлы `CopyToReg`, `CopyFromReg` и `Register` остались нетронутыми, и будут оставаться такими до этапа распределения регистров. Фактически, на этапе выбора инструкций могут быть даже сгенерированы новые узлы. После выбора инструкций узел `ISD::ADD` превратился в инструкцию `ADD32ri8`, а `X86ISD::RET_FLAG` в `RET`.

Примечание. Отметьте, что в одном графе DAG могут мирно уживаться три типа представлений инструкций: обобщенные узлы `ISD`, такие как `ISD::ADD`, архитектурно-зависимые узлы `<Целевая_архитектура>ISD`, такие как `X86ISD::RET_FLAG`, и физические машинные инструкции, такие как `X86::ADD32ri8`.

Сопоставление с шаблоном

Поддержка каждой целевой архитектуры реализует свой метод `Select` выбора инструкций в подклассе `<Целевая_архитектура>DAGToDAGISel`, наследующем класс `SelectionDAGISel`, например, поддержка архитектуры SPARC реализует метод `SparcDAGToDAGISel::Select()` (см. `lib/Target/Sparc/SparcISelDAGToDAG.cpp`). Этот метод принимает узел `SDNode` для сопоставления и возвращает значение `SDNode`, представляющее физическую инструкцию.

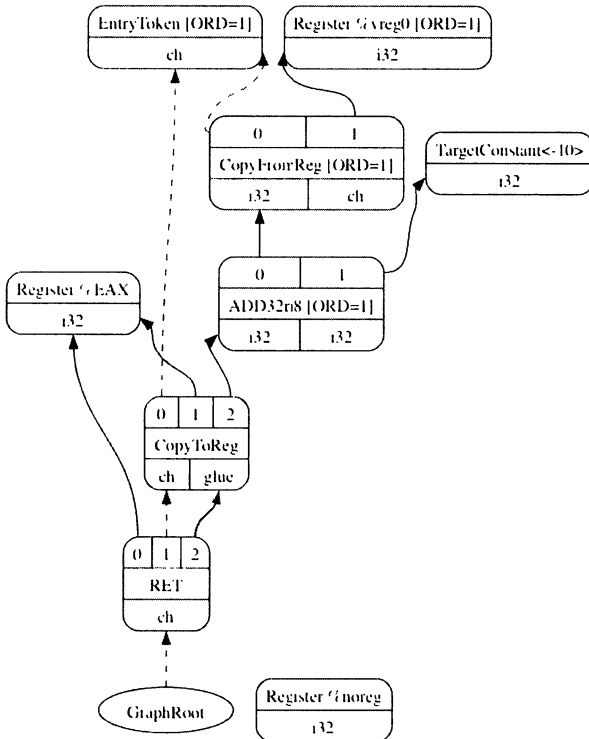


Рис. 6.4. Структура `SelectionDAG` после этапа выбора инструкций

Метод `Select()` предусматривает два способа сопоставления. Наиболее прямолинейный заключается в использовании кода сопоставления, сгенерированного из шаблонов `TableGen`, как описывается в п. 1, в списке ниже. Однако шаблоны могут оказаться недостаточно выразительными, чтобы охватить все тонкости некоторых

инструкций. Для таких случаев в методе должна быть реализована собственная, нестандартная логика сопоставления, как описывается в п. 2 ниже. Далее оба способа описываются подробнее:

1. Метод `Select()` вызывает `SelectCode()`, который генерируется на основе определений `TableGen` для каждой целевой архитектуры. Кроме этого метода генерируется также таблица соответствий `MatcherTable`, отображающая узлы `ISD` и `<Целевая_архитектура>ISD` в узлы физических инструкций. Таблица соответствий генерируется на основе определений в файлах `.td` (обычно `<Целевая_архитектура>InstrInfo.td`). Метод `SelectCode()` завершается вызовом `SelectCodeCommon()`, архитектурно-независимого метода сопоставления узлов с использованием целевой таблицы соответствий. В поддержке `TableGen` имеется специальный генератор кода выбора инструкций, который создает все эти методы и таблицу соответствий:

```
$ cd <llvm_source>/lib/Target/Sparc
$ llvm-tblgen -gen-dag-isel Sparc.td -I ../../../../include
```

Для каждой целевой архитектуры генерируется один и тот код на языке `C++` в файле `<каталог_сборки>/lib/Target/<Целевая_архитектура>/<Целевая_архитектура>GenDAGISel.inc`; например, для `SPARC` методы и таблица доступны в `<каталог_сборки>/lib/Target/Sparc/SparcGenDAGISel.inc`.

2. В метод `Select`, перед вызовом `SelectCode`, можно добавить собственный код. Например, узел `ISD::MULHU` типа `i32` описывает операцию умножения двух значений `i32`, результат которой имеет тип `i64`, и возвращает старшее слово `i32` результата. В 32-разрядной архитектуре `SPARC` инструкция умножения `SP::UMULrr` возвращает старшее слово в специальном регистре `Y`, для чтения которого должна использоваться специальная инструкция `SP::RDY`. Механизм `TableGen` неспособен воспроизвести эту логику, но мы легко можем сделать это сами, добавив следующий код:

```
case ISD::MULHU: {
    SDValue MulLHS = N->getOperand(0);
    SDValue MulRHS = N->getOperand(1);
    SDNode *Mul = CurDAG->getMachineNode(SP::UMULrr, dl,
        MVT::i32, MVT::Glue, MulLHS, MulRHS);
    return CurDAG->SelectNodeTo(N, SP::RDY, MVT::i32,
```



```

SDValue (Mul, 1));
}

```

Здесь `N` – это аргумент типа `SDNode` и в данном контексте представляет инструкцию `ISD::MULHU`. Так как все необходимые проверки на допустимость были выполнены до этой инструкции `case`, мы сразу переходим к преобразованию инструкции `ISD::MULHU` в операции для архитектуры `SPARC`. Для этого вызовом `CurDAG->getMachineNode()` создается узел с физической инструкцией `SP::UMULrr`. Далее, с помощью `CurDAG->SelectNodeTo()` создается узел инструкции `SP::RDY` и все обращения к результату `ISD::MULHU` замещаются обращениями к результату `SP::RDY`. На рис. 6.5 изображена диаграмма структуры `SelectionDAG` для данного примера до и после выбора инструкций. Предыдущий фрагмент кода на C++ – это упрощенная версия из файла `lib/Target/Sparc/SparcISelDAGToDAG.cpp`.

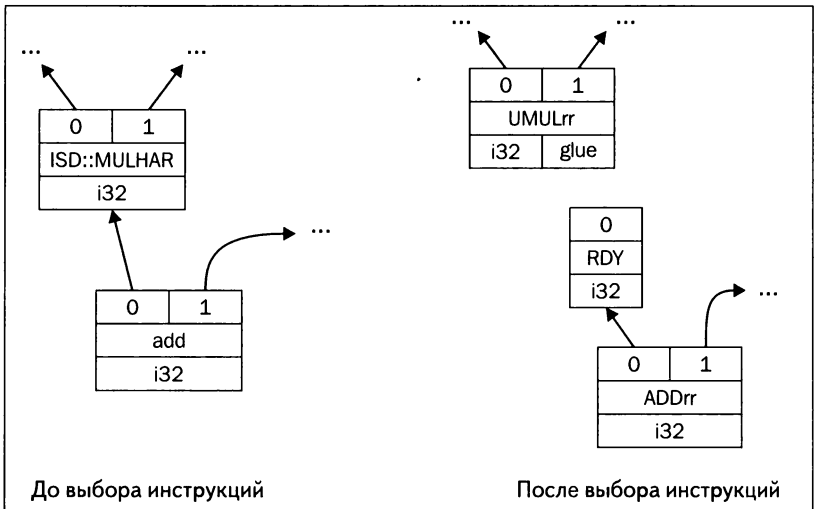


Рис. 6.5. Структура `SelectionDAG` до и после выбора инструкций

Визуализация процесса выбора инструкций

Инструмент `llc` имеет несколько параметров, позволяющих получить визуальное представление графа `SelectionDAG` на разных эта-

нах выбора инструкций. Если указать один из этих параметров, llc сгенерирует графический файл .dot с изображением, подобным тем, что были показаны выше в этой главе, но вам потребуется использовать программу dot, чтобы вывести такой файл на экран, или dotty, чтобы отредактировать его. Обе программы можно найти в пакете Graphviz (<http://www.graphviz.org>). В табл. 6.2 перечислены все параметры в порядке выполнения.

Таблица 6.2. Параметры визуализации процесса выбора инструкций

Параметр для llc	Этап
-view-dag-combine1-dags	Перед первым объединением DAG
-view-legalize-types-dags	Перед легализацией типов
-view-dag-combine-1t-dags	После второй легализации типов и перед объединением DAG
-view-legalize-dags	Перед легализацией
-view-dag-combine2-dags	Перед вторым объединением DAG
-view-isel-dags	Перед выбором инструкций
-view-sched-dags	После выбора инструкций и перед планированием

Быстрый выбор инструкций

LLVM поддерживает также альтернативную реализацию выбора инструкций, которую называют *быстрым выбором инструкций* (в виде класса FastISel, см. файл `<llvm_source>/lib/CodeGen/SelectionDAG/FastISel.cpp`). Цель этой альтернативы – ускорить создание кода за счет его качества, в соответствии с философией уровня оптимизации -O0. Ускорение достигается за счет исключения из процесса сложной логики свертки и упрощения. Для простых операций так же используются описания на языке TableGen, а более сложные обрабатываются архитектурно-зависимым кодом.

Примечание. На уровне оптимизации -O0 также используются быстрые, но неоптимальные реализации распределения регистров и планирования инструкций. Мы рассмотрим их в следующих разделах.

Планирование инструкций

После выбора инструкций в структуре `SelectionDAG` остаются узлы, представляющие физические инструкции, напрямую поддерживаемые целевой архитектурой. Следующий этап – планирование узлов `SelectionDAG` (`SDNodes`) перед этапом распределения регистров. Существует несколько планировщиков на выбор и все они являются подклассами класса `ScheduleDAGSDNodes` (см. `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGSDNodes.cpp`). Тип планировщика можно выбирать с помощью параметра `-pre-RA-sched=<планировщик>` инструмента `llc`. Ниже перечислены возможные значения для `<планировщик>`:

- `list-ilp`, `list-hybrid`, `source` и `list-burr`: эти значения определяют списки алгоритмов планирования, реализованных в классе `ScheduleDAGRRLList` (см. `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGRRLList.cpp`);
- `fast`: класс `ScheduleDAGFast` (в `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGFast.cpp`), реализующий неоптимальный, но быстрый алгоритм планирования;
- `vliw-td`: Планировщик для архитектуры VLIW, реализованный в виде класса `ScheduleDAGVLIW` (см. `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGVLIW.cpp`).

Значение `default` выбирает лучший из predeterminedных планировщиков для заданной архитектуры, а значение `linearize` позволяет вообще запретить планирование. Для оптимального планирования доступные планировщики могут использовать информацию из маршрутов инструкций и возвращаемую механизмами определения опасностей.

Примечание. В генераторе кода используется три разных планировщика: два вызываются перед распределением регистров, и один – после. Первый работает с узлами `SelectionDAG`, а другие два – с машинными инструкциями, как описывается далее в этой главе.

Маршруты инструкций

Некоторые целевые архитектуры предоставляют маршруты инструкций – информацию о продолжительности выполнения инструкций

и об аппаратном конвейсере. Планировщик использует эти данные в процессе планирования, чтобы обеспечить максимальную пропускную способность и производительность. Эта информация описывается в файлах на языке TableGen, обычно с именами <Целевая_архитектура>Schedule.td (например, X86Schedule.td).

В LLVM имеется TableGen-класс ProcessorItineraries, в файле <llvm_source>/include/llvm/Target/TargetItinerary.td:

```
class ProcessorItineraries<list<FuncUnit> fu, list<Bypass> bp,
                          list<InstrItinData> iid> {
  ...
}
```

Целевые архитектуры могут определять собственные обработчики маршрутов. Для этого следует определить список функциональных блоков (FuncUnit), байпас-шин¹ (pipeline bypasses, Bypass) и данные о маршруте инструкции (InstrItinData). Например, в файле <llvm_source>/lib/Target/ARM/ARMScheduleA8.td определен маршрут для инструкций ARM Cortex A8:

```
def CortexA8Itineraries : ProcessorItineraries<
  [A8_Pipe0, A8_Pipe1, A8_LSPipe, A8_NPipe, A8_NLSPipe],
  [], [
  ...
  InstrItinData<IIC_iALUi, [InstrStage<1, [A8_Pipe0, A8_Pipe1]>],
  [2, 2]>,
  ...
]>;
```

Здесь указывается, что байпас-шины отсутствуют. Приводится список функциональных блоков (A8_Pipe0, A8_Pipe1 и другие) данного процессора и информация о маршрутах инструкций типа IIC_iALUi. Этот тип является классом двухместных инструкций в форме `reg = reg + immediate`, таких как инструкции ADDri и SUBri. Эти инструкции выполняются за один машинный такт и задействуют функциональные блоки A8_Pipe0 и A8_Pipe1, согласно определению в InstrStage<1, [A8_Pipe0, A8_Pipe1]>.

Следующий далее список [2, 2] представляет число циклов после запуска инструкции, необходимых для чтения или определения операндов. В данном случае регистр назначения (с индексом 0) и регистр с исходными данными (с индексом 1) становятся доступны через 2 такта.

¹ <https://ru.wikipedia.org/wiki/Байпас>. – Прим. перев.

Определение опасностей

Механизм определения опасностей (*hazard recognizer*) опирается на информацию из маршрутов. Класс *ScheduleHazardRecognizer* предоставляет интерфейс к реализациям механизмов определения опасностей, а его подкласс *ScoreboardHazardRecognizer* реализует механизм определения опасностей (см. файл `<llvm_source>/lib/CodeGen/ScoreboardHazardRecognizer.cpp`), который используется в LLVM по умолчанию.

Целевые архитектуры могут предоставлять собственные реализации таких механизмов. Это может потребоваться, потому что язык *TableGen* не всегда способен выразить специфические ограничения, которые необходимо реализовать вручную. Например, обе архитектуры – ARM и PowerPC – реализуют собственные классы, наследующие *ScoreboardHazardRecognizer*.

Единицы планирования

Планировщик запускается до и после распределения регистров. Однако, представление инструкций в виде объектов *SDNode* доступно только в первом случае, во втором инструкции представлены экземплярами класса *MachineInstr*. Чтобы иметь возможность обрабатывать оба представления инструкций – *SDNode* и *MachineInstr* – доступ к ним на этапе планирования осуществляется посредством класса *SUnit* (см. `<llvm_source>/include/llvm/CodeGen/ScheduleDAG.h`), представляющего единицу планирования. Инструмент *llc* позволяет вывести граф единиц планирования, для чего достаточно передать ему ключ `-view-sunit-dags`.

Машинные инструкции

Механизм распределения регистров работает с инструкциями, представленными экземплярами класса *MachineInstr* (MI, для краткости), определение которого находится в файле `<llvm_source>/include/llvm/CodeGen/MachineInstr.h`. Проход *InstrEmitter*, выполняющийся после планирования, преобразует инструкции из формата *SDNode* в формат *MachineInstr*. Как можно догадаться по имени, это представление ближе к фактическому машинному представлению для заданной архитектуры, чем представление IR. В отличие от формата *SDNode*, формат MI является трехадресным представлением программы, когда программа имеет вид последовательности

инструкций, а не ориентированного графа DAG, что позволяет компилятору эффективно принимать конкретные решения на этапе планирования. Каждый экземпляр MI хранит код операции – число, имеющее смысл только для определенного генератора кода, – и список операндов.

Вызвав `llc` с ключом `-print-machineinstrs`, можно вывести дамп машинных инструкций после всех зарегистрированных проходов, а если передать ключ с параметром: `-print-machineinstrs=<имя-прохода>`, `llc` выведет дамп после указанного прохода. Имена проходов можно найти в исходных текстах LLVM. Для этого перейдите в каталог с исходными текстами LLVM и с помощью `grep` выполните поиск макроса, который обычно используется для регистрации имен проходов:

```
$ grep -r INITIALIZE_PASS_BEGIN *
CodeGen/PHIElimination.cpp:INITIALIZE_PASS_BEGIN(PHIElimination,
"phinode-elimination"
(...)
```

В качестве примера ниже приводится дамп машинных инструкций SPARC для `sum.bc`, полученный после всех проходов:

```
$ llc -march=sparc -print-machineinstrs sum.bc

Function Live Ins: %I0 in %vreg0, %I1 in %vreg1
BB#0: derived from LLVM BB %entry
  Live Ins: %I0 %I1
  %vreg1<def> = COPY %I1; IntRegs:%vreg1
  %vreg0<def> = COPY %I0; IntRegs:%vreg0
  %vreg2<def> = ADDrr %vreg1, %vreg0; IntRegs:%vreg2,%vreg1,%vreg0
  %I0<def> = COPY %vreg2; IntRegs:%vreg2
  RETL 8, %I0<imp-use>
```

Экземпляры MI содержат важную метаинформацию об инструкциях: используемые и определяемые регистры (различает операнды в регистрах и в памяти, и др.), тип инструкции (ветвление, возврат, вызов, завершение и др.), предикаты, например, определяющие коммутативность, и так далее. Эта информация играет важную роль даже на самых низких уровнях, таких как уровень MI, потому что проходы, выполняемые после `InstrEmitter` и перед эмиссией кода, используют ее для выполнения собственных видов анализа.

Распределение регистров

Основной целью этапа распределения регистров является преобразование бесконечного числа виртуальных регистров в ограничен-

ное множество физических регистров. Целевые архитектуры имеют ограниченное число физических регистров, поэтому некоторым виртуальным регистрам ставятся в соответствие ячейки памяти, *сверхнормативные слоты* (spill slots). И все же, некоторые фрагменты кода в представлении MI могут использовать физические регистры еще до этапа распределения регистров. Это, например, характерно для машинных инструкций, которые должны записывать результаты в какие-то определенные регистры, или может быть обусловлено требованиями ABI. В таких случаях механизм распределения регистров учитывает занятые регистры и для связывания виртуальных регистров использует оставшиеся физические регистры.

Еще одной важной функцией распределения регистров в LLVM является преобразование формата SSA промежуточного представления IR. До настоящего момента машинные инструкции могут содержать фи-инструкции (phi instructions), скопированные из оригинального кода LLVM IR и необходимые для поддержки формата SSA. Благодаря этому есть возможность реализовать архитектурно-зависимые оптимизации с учетом SSA. Однако, традиционно фи-инструкции преобразуются в обычные инструкции путем замены их инструкциями копирования. То есть, преобразование формата не должно откладываться на этапы, следующие за этапом распределения регистров, на котором осуществляется связывание регистров и устранение избыточных операций копирования.

В LLVM имеется четыре реализации механизма распределения регистров, которые можно выбирать при вызове инструмента `llc` с помощью параметра `-regalloc=<имя_механизма>`. Где `<имя_механизма>` может принимать одно из следующих значений: `pbqp`, `greedy`, `basic` и `fast`.

- `pbqp`: этот механизм отображает задачу распределения регистров в задачу **секционированного булева квадратичного программирования (Partitioned Boolean Quadratic Programming, PBQP)**. Решение PBQP используется для обратного отображения результата в регистры.
- `greedy`: этот механизм реализует эффективное глобальное (в пределах функций) распределение регистров с учетом времени жизни значений (live-range splitting), чтобы уменьшить использование памяти до минимума. Превосходное описание этого алгоритма можно найти по адресу: <http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html>.

- *basic*: этот механизм использует очень простой алгоритм распределения и предоставляет интерфейс для подключения расширений. То есть, он обеспечивает основу для разработки новых механизмов распределения регистров. Дополнительную информацию об этом алгоритме можно найти в той же статье с описанием механизма *greedy*, адрес которой приводился выше.
- *fast*: это механизм локального распределения регистров (действующий на уровне базовых блоков). Он стремится хранить все значения в регистрах и использовать их повторно всегда, когда это возможно.

Механизм распределения по умолчанию выбирается, в зависимости от текущего уровня оптимизации (параметра `-O`).

Даже при том, что распределение регистров (независимо от алгоритма) выполняется в единственном проходе, оно все еще зависит от других проходов анализа, составляющих инфраструктуру распределения. Существует несколько таких проходов и, чтобы описать суть концепции, мы расскажем вам о проходах объединения регистров и замены виртуальных регистров. На рис. 6.6 показано, как взаимодействуют эти проходы.

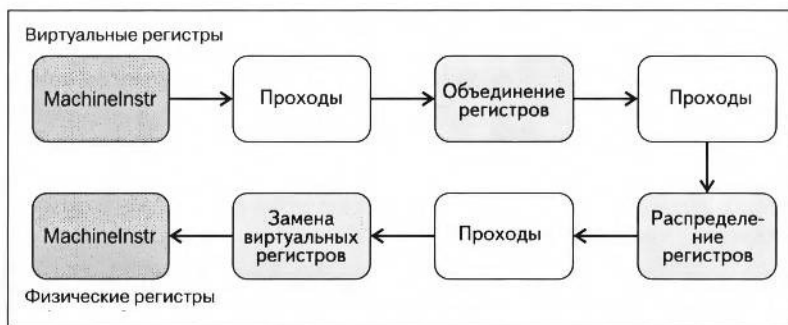


Рис. 6.6. Диаграмма взаимодействий проходов в инфраструктуре распределения регистров

Объединение регистров

На проходе объединения регистров удаляются избыточные инструкции копирования (`COPY`) за счет объединения интервалов. Объединение реализовано в виде класса `RegisterCoalescer` (см. `lib/`

CodeGen/RegisterCoalescer.cpp) – действующего на уровне машинной функции. Проход на уровне машинной функции напоминает проход, действующий на уровне функций IR, только вместо инструкций промежуточного представления обрабатывает инструкции в виде экземпляров MachineInstr. В ходе объединения метод joinAllIntervals() выполняет обход списка инструкций копирования. Метод joinCopy() создает экземпляры CoalescerPair из машинных инструкций копирования и объединяет их, когда это возможно.

Интервал – это пара точек в программе, начало и конец, которые отмечают момент создания значения и момент последнего его использования. Давайте посмотрим, как будет выполнено объединение в файле sum.bc.

Получить результаты объединения можно, если передать инструменту llc отладочный параметр regalloc:

```
$ llc -march=sparc -debug-only=regalloc sum.bc 2>&1 | head -n30
```

```
Computing live-in reg-units in ABI blocks.
```

```
0B BB#0 I0#0 I1#0
```

```
***** INTERVALS *****
```

```
I0 [0B,32r:0) [112r,128r:1) 0@0B-phi 1@112r
```

```
I1 [0B,16r:0) 0@0B-phi
```

```
%vreg0 [32r,48r:0) 0@32r
```

```
%vreg1 [16r,96r:0) 0@16r
```

```
%vreg2 [80r,96r:0) 0@80r
```

```
%vreg3 [96r,112r:0) 0@96r
```

```
RegMasks:
```

```
***** MACHINEINSTRS *****
```

```
# Machine code for function sum: Post SSA
```

```
Frame Objects:
```

```
fi#0: size=4, align=4, at location[SP]
```

```
fi#1: size=4, align=4, at location[SP]
```

```
Function Live Ins: $I0 in %vreg0, $I1 in %vreg1
```

```
0B BB#0: derived from LLVM BB %entry
```

```
Live Ins: %I0 %I1
```

```
16B          %vreg1<def> = COPY %I1<kill>; IntRegs:%vreg1
```

```
32B          %vreg0<def> = COPY %I0<kill>; IntRegs:%vreg0
```

```
48B          STri <fi#0>, 0, %vreg0<kill>; mem:ST4[%a.addr]
```

```
IntRegs:%vreg0
```

```
64B      STri <fi#1>, 0, %vreg1; mem:ST4[%b.addr]
IntRegs:$vreg1
80B      %vreg2<def> = LDri <fi#0>, 0; mem:LD4[%a.addr]
IntRegs:%vreg2
96B      %vreg3<def> = ADDrr %vreg2<kill>, %vreg1<kill>;
IntRegs:%vreg3,%vreg2,%vreg1
112B     %I0<def> = COPY %vreg3<kill>; IntRegs:%vreg3
128B     RETL 8, %I0<imp-use,kill>
```

```
# End machine code for function sum.
```

Примечание. Существует возможность включить вывод отладочных сообщений для определенного прохода или компонента LLVM передачей параметра `-debug-only`. Чтобы найти компоненты для отладки, выполните команду `grep -r "DEBUG_TYPE" *` в дереве каталогов с исходными текстами LLVM. Макрос `DEBUG_TYPE` определяет значение параметра, активирующее вывод отладочных сообщений, например, в файлах реализации распределения регистров используется: `#define DEBUG_TYPE "regalloc"`.

Обратите внимание, что конструкция `2>&1` в примере выше перенаправляет стандартный вывод ошибок, куда обычно выводится отладочная информация, на стандартный вывод. Далее стандартный вывод (то есть, вся отладочная информация) по конвейеру передается команде `head -n30`, чтобы вывести только первые 30 строк. Таким способом можно управлять объемом информации, выводимой на экран, потому что иногда этой информации может оказаться очень много.

Для начала рассмотрим вывод, начиная со строки `** MACHINEINSTRS **`. Это дамп всех машинных инструкций, попадающих на вход прохода объединения регистров. То же самое можно получить, вызвав `llc` с параметром `-print-machine-insts=phi-node-elimination`, который управляет выводом машинных инструкций после прохода устранения фи-узлов (этот проход производится перед проходом объединения регистров), с той лишь разницей, что в отладочной информации машинные инструкции снабжены индексами для каждого экземпляра MI: 0B, 16B, 32B и так далее. Они потребуются нам для правильной интерпретации интервалов.

Эти индексы также называют *индексами слотов* (slot indexes) и присваивают разные номера всем периодам жизни значений. Буква `v` означает «Block» (блок) и используется для обозначения точек начала/конца периодов в пределах базового блока. В данном случае индексы завершаются буквой `v`, потому что это слот по умолчанию.

Другой слот, обозначаемый буквой *r*, можно увидеть в разделе `*** INTERVALS ***` со списком интервалов. Здесь буква *r* означает «register» (регистр) и указывает, что это слот использования/определения обычных регистров.

После чтения списка машинных инструкций мы уже знаем, что наиболее важные аспекты суперпрохода распределения регистров (состоящего из нескольких проходов поменьше): виртуальные регистры `%vreg0`, `%vreg1`, `%vreg2` и `%vreg3` требуется распределить по физическим регистрам. То есть, всего понадобится не более четырех физических регистров, помимо `%I0` и `%I1`, которые уже заняты в соответствии с соглашениями ABI, требующими передавать параметры функций в этих регистрах. Так как определение периодов жизни переменных выполняется перед объединением, код дополнительно сопровождает информация об этих периодах, указывающая, в какие моменты каждый регистр начинает и прекращает использоваться, что дает возможность увидеть, периоды жизни каких регистров пересекаются и их нельзя распределить в один и тот же физический регистр.

Не зависящий от результатов распределения регистров, механизм объединения, с другой стороны, просто ищет операции копирования регистров. Найдя операцию копирования из регистра в регистр, он пытается объединить интервалы использования двух регистров и отобразить их в один и тот же физический регистр, чтобы избавиться от инструкций копирования, подобных инструкциям с индексами 16 и 32.

Первые строки в разделе `*** INTERVALS ***` получены в ходе выполнения другого анализа, от которого зависит проход объединения регистров: анализа определения интервалов использования (отличается от анализа интервалов использования переменных), реализованного в `lib/CodeGen/LiveIntervalAnalysis.cpp`. Механизм объединения должен знать интервалы, где используется каждый виртуальный регистр, чтобы решить, какие интервалы объединить. Например, из вывода выше, видно, что для виртуального регистра `%vreg0` определен интервал `[32r:48r:0)`.

Это обозначение описывает полуоткрытый интервал, где регистр `%vreg0` определен в инструкции с индексом 32 и последний раз использовался в инструкции с индексом 48. Число 0, следующее за `48r`, — это код, показывающий, где находится первое определение этого интервала. Значение кода выводится сразу за интервалом: `0@32r`. То есть, определение 0 в индексе 32, о чем мы и так знаем. Однако, эта информация может пригодиться, если потребуется отследить место определения, когда интервалы разбиты. Наконец, `RegMasks` показы-

васт точки вызова, где затирается большое число регистров. Поскольку наша функция не вызывает никаких других функций, в RegMask не указано ничего.

После выявления всех интервалов можно определить наиболее перспективные: регистр %I0 имеет интервал [0B, 32r:0), регистр %vreg0 имеет интервал [32r, 48r:0) и в точке 32 выполняется копирование содержимого %I0 в %vreg0. На лицо все предпосылки к объединению, поэтому вполне можно объединить интервалы [0B, 32r:0) и [32r:48r:0), и связать регистры %I0 и %vreg0 с одним и тем же физическим регистром.

Теперь посмотрим остаток отладочного вывода и выясним, что произошло:

```
$ llc -march=sparc -debug-only=regalloc sum.bc
...
entry:
16B %vreg1<def> = COPY %I1; IntRegs:%vreg1
    Considering merging %vreg1 with %I1
    Can only merge into reserved registers.
    (Проверяется возможность объединения %vreg1 и %I1
    Объединение возможно только в зарезервированные регистры.)
32B %vreg0<def> = COPY %I0; IntRegs:%vreg0
    Considering merging %vreg0 with %I0
    Can only merge into reserved registers.
    (Проверяется возможность объединения %vreg0 и %I0
    Объединение возможно только в зарезервированные регистры.)
64B %I0<def> = COPY %vreg2; IntRegs:%vreg2
    Considering merging %vreg2 with %I0
    Can only merge into reserved registers.
    (Проверяется возможность объединения %vreg2 и %I0
    Объединение возможно только в зарезервированные регистры.)
....
```

Как видите, возможность объединения регистров %vreg0 и %I0 была обнаружена, как мы и хотели. Однако, когда один из регистров является физическим регистром, таким как %I0, в силу вступает специальное правило. Физический регистр должен быть *зарезервирован*, чтобы к его интервалу можно было присоединить другой интервал. Это означает, что физический регистр не должен быть доступен для распределения в других интервалах, что совсем не так в случае с регистром %I0. Далее механизм объединения отвергает возможность объединения из-за риска, что преждевременное связывание %I0 с этим интервалом будет невыгодно в долгосрочном плане, и оставляет это решение за механизмом распределения регистров.

То есть, программа `sum.bc` не представила возможности объединения. Несмотря на обнаружившуюся возможность объединения виртуальных регистров с регистрами, в которых передаются параметры функции, попытка объединения потерпела неудачу, потому что на этом этапе виртуальные регистры могут объединяться только с зарезервированными физическими регистрами, недоступными для обычного распределения.

Замена виртуальных регистров

Проход распределения регистров выбирает физический регистр для каждого виртуального регистра. Результат распределения в виде отображения виртуальных регистров в физические сохраняется затем в `VirtRegMap`. Далее выполняется проход замены виртуальных регистров, реализованный в виде класса `VirtRegRewriter` в файле `<llvm_source>/lib/CodeGen/VirtRegMap.cpp`, который использует `VirtRegMap` и замещает виртуальные регистры физическими. При необходимости генерируется дополнительный код. Кроме того, на этом проходе уничтожаются оставшиеся тождественные операции `reg = COPY reg`. Например, давайте посмотрим на результаты распределения и замены регистров в файле `sum.bc`, воспользовавшись для этого параметром `-debug-only=regalloc`. Для начала взгляните на результаты, произведенные механизмом распределения `greedy`:

```
...
assigning %vreg1 to %I1: I1
...
assigning %vreg0 to %I0: I0
...
assigning %vreg2 to %I0: I0
```

Виртуальные регистры 1, 0 и 2 отображены в физические регистры `%I1`, `%I0` и `%I0`, соответственно. Те же результаты в `VirtRegMap` имеют следующий вид:

```
{%vreg0 -> %I0} IntRegs
{%vreg1 -> %I1} IntRegs
{%vreg2 -> %I0} IntRegs
```

Далее все виртуальные регистры заменяются физическими и производится удаление тождественных операций копирования:

```
> %I1<def> = COPY %I1
Deleting identity copy.

> %I0<def> = COPY %I0
Deleting identity copy.
```

Как видите, даже при том, что проход объединения не смог удалить эту операцию копирования, механизм распределения регистров смог связать один и тот же регистр с двумя диапазонами и удалить избыточную операцию. Как результат, число машинных инструкций, составляющих функцию `sum`, существенно уменьшилось:

```
0B BB#0: derived from LLVM BB %entry
  Live Ins: %I0 %I1

48B %I0<def> = ADDrr %I1<kill>, %I0<kill>

80B RETL 8, %I0<imp-use>
```

Обратите внимание, что все инструкции копирования были удалены и не осталось ни одного виртуального регистра.

Примечание. Параметры `-debug` и `-debug-only=<имя>` инструмента `llc` доступны, только когда проект LLVM скомпилирован с отладочной информацией, с включенным параметром `--disable-optimized` во время конфигурирования. Более подробная информация о порядке настройки, сборки и установки приводится в разделе «Сборка и установка LLVM», в главе 1, «Сборка и установка LLVM».

Механизмы распределения регистров и планирования инструкций известные антагонисты в любом компиляторе. Задача механизма распределения регистров состоит в том, чтобы сделать диапазоны как можно более короткими, за счет уменьшения числа ребер в графе и, как следствие, уменьшить число необходимых регистров, чтобы избежать использования памяти. Поэтому для него предпочтительнее, когда инструкции планируются последовательно (инструкции, зависящие друг от друга, следуют непрерывной чередой), потому что такой код использует меньшее число регистров. Задача механизма планирования полностью противоположна: обеспечить параллельное выполнение инструкций, то есть параллельное выполнение множества несвязанных вычислений, что требует большого числа регистров для хранения промежуточных результатов и увеличивает вероятность пересечения интервалов использования значений. Создание эффективного алгоритма, который примирил бы механизмы планирования и распределения до сих пор остается неразрешимой задачей.

Архитектурно-зависимые обработчики

На этап объединения виртуальные регистры должны поступать в виде экземпляров совместимых классов регистров. Генератор кода извлекает информацию о типах из архитектурно-зависимого описания, используя для этого абстрактные методы. Механизм распреде-

ления регистров может получить всю необходимую информацию о регистрах с помощью подклассов, наследующих `TargetRegisterInfo` (например, `X86GenRegisterInfo`); в том числе признак зарезервированного регистра, родительские классы регистра и его принадлежность к категории физических или виртуальных регистров.

Класс `<Целевая_архитектура>InstrInfo` – это еще одна структура данных, являющаяся источником архитектурно-зависимой информации, необходимой для распределения. Ниже приводится описание некоторых его методов.

Методы `isLoadFromStackSlot()` и `isStoreToStackSlot()` используются для определения, является ли машинная инструкция обращением к слоту памяти на стеке.

Кроме того, генератор кода создает архитектурно-зависимые инструкции обращения к стеку, используя методы `storeRegToStackSlot()` и `loadRegFromStackSlot()`.

По окончании этапа замены в коде еще могут оставаться инструкции `COPY`, из-за того, что не были объединены и не выполняют тождественное копирование. В таких случаях вызывается метод `copyPhysReg()`, чтобы сгенерировать архитектурно-зависимую операцию копирования регистров, даже при том, что регистры могут принадлежать к разным классам. Ниже приводится фрагмент кода из `SparcInstrInfo::copyPhysReg()`:

```
if (SP::IntRegsRegClass.contains(DestReg, SrcReg))
    BuildMI(MBB, I, DL, get(SP::ORrr), DestReg).addReg(SP::G0)
        .addReg(SrcReg, getKillRegState(KillSrc));
...
```

Метод `BuildMI()` используется всегда, когда генератору требуется сгенерировать машинную инструкцию. В этом примере для копирования содержимого одного машинного регистра в другой используется инструкция `SP::ORrr`.

Пролог и эпилог

Каждая законченная функция должна иметь пролог и эпилог. Пролог выполняет подготовку кадра стека и сохраняет содержимое регистров вызывающей программы в начале функции, а эпилог освобождает кадр стека перед возвратом из функции. Ниже показано, как выглядят инструкции, составляющие пролог и эпилог в нашем примере с файлом `sum.bc`, когда компиляция выполняется для архитектуры SPARC:

```
%O6<def> = SAVErri %O6, -96
%I0<def> = ADDRr %I1<kill>, %I0<kill>
%G0<def> = RESTORErr %G0, %G0
RETL 8, %I0<imp-use>
```

Здесь инструкция `SAVErri` – это пролог, а `RESTORErr` – эпилог. Эти инструкции выполняют операции по подготовке и освобождению кадра стека. Машинный код, реализующий пролог и эпилог, зависит от конкретной архитектуры и определяется в методах `<Целевая_архитектура>FrameLowering::emitPrologue()` и `<Целевая_архитектура>FrameLowering::emitEpilogue()` (см. файл `<llvm_source>/lib/Target/<Целевая_архитектура>/<Целевая_архитектура>FrameLowering.cpp`).

Индексы кадров стека

В процессе создания выполняемого кода LLVM использует виртуальные кадры стека и ссылается на элементы стека с помощью индексов кадров. При добавлении пролога размещается кадр стека и возвращается достаточный объем информации, чтобы генератор кода смог заменить индексы в виртуальном кадре фактическими (архитектурно-зависимыми) ссылками на память в стеке.

Метод `eliminateFrameIndex()` класса `<Целевая_архитектура>RegisterInfo` реализует такую замену, преобразуя каждый индекс в кадре действительным смещением в стеке во всех машинных инструкциях, содержащих ссылки на стек (обычно инструкции чтения из стека и записи в стек). При необходимости генерируются дополнительные инструкции, выполняющие арифметические операции со смещениями в стеке. Примеры можно найти в файле `<llvm_source>/lib/Target/<Целевая_архитектура>/<Целевая_архитектура>RegisterInfo.cpp`.

Инфраструктура машинного кода

Классы, представляющие **машинный код** (MC, для краткости), образуют полноценную инфраструктуру для управления низкоуровневыми функциями и инструкциями. В сравнении с другими компонентами генератора кода, данная инфраструктура была создана относительно недавно, чтобы помочь разработчикам в создании ассемблеров и дизассемблеров на основе LLVM. Прежде в LLVM отсутствовал встроенный ассемблер, и компиляцию можно было выпол-

нить только до этапа эмиссии ассемблерного кода, то есть до создания файла сборки, и затем использовать внешние инструменты (ассемблеры и компоновщики), для выполнения оставшейся части процедуры компиляции.

Инструкции MC

Инфраструктура MC замещает машинные инструкции (`MachineInstr`) машинным кодом (`MCInst`). Класс `MCInst` (определяется в `<llvm_source>/include/llvm/MC/MCInst.h`) является легковесным представлением инструкций. В сравнении с экземплярами `MI`, экземпляры `MCInst` несут меньше информации о программе. Например, экземпляр `MCInst` может быть создан не только генератором кода, но также дизассемблером из двоичного кода, в окружении, где не так много информации о контексте инструкции. Фактически, `MCInst` является представлением уровня ассемблера – инструмента, цель которого состоит не в применении оптимизаций, а в организации инструкций в объектный файл.

Операндами могут быть регистры, непосредственные значения (целые или вещественные числа), выражения (в виде экземпляров `MSExp`) или другие экземпляры `MCInstr`. Выражения используются для представления меток и смещений. Инструкции `MI` преобразуются в экземпляры `MCInst` на ранних стадиях этапа эмиссии кода, о котором рассказывается в следующем подразделе.

Эмиссия кода

Этап эмиссии кода выполняется после всех проходов распределения регистров. Этот этап, с таким странным названием, начинается с прохода вывода сборки (`AsmPrinter`). На рис. 6.7 показаны все шаги по преобразованию инструкций `MI` в инструкции `MCInst` и далее в ассемблерный или двоичный машинный код.

Рассмотрим по очереди все шаги, представленные на диаграмме (рис. 6.7):

1. `AsmPrinter` – это проход обработки машинных функций, который сначала выводит заголовок функции, а затем выполняет итерации по всем базовым блокам, передавая инструкции `MI` по одной методу `EmitInstruction()` для дальнейшей обработки. Для каждой целевой архитектуры определяется свой подкласс класса `AsmPrinter`, в котором переопределяется этот метод.

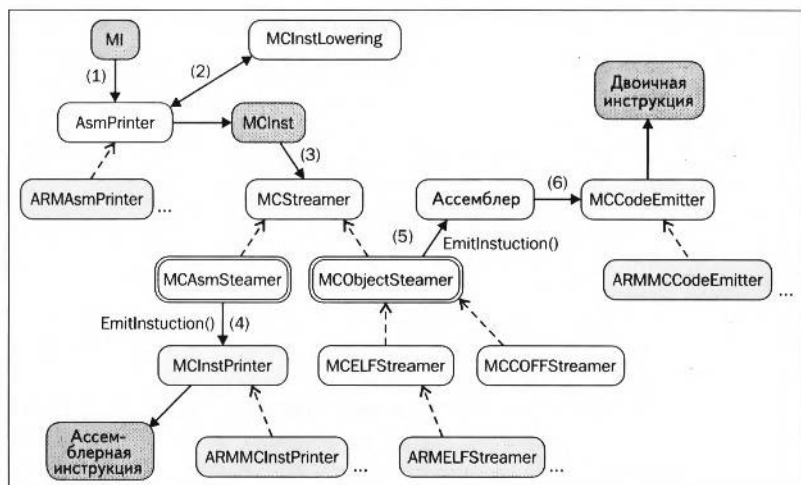


Рис. 6.7. Диаграмма преобразования инструкций MI в инструкции MCInst и далее в ассемблерный или двоичный машинный код

- Метод `<Целевая_архитектура>AsmPrinter::EmitInstruction()` принимает инструкцию MI и преобразует ее в экземпляр MCInst с помощью интерфейса MCInstLowering – для каждой целевой архитектуры определяется свой подкласс этого интерфейса, генерирующий экземпляры MCInst.
- Дальше работа может пойти по одному из двух направлений: эмиссия ассемблерного кода или двоичных инструкций. класс MCStreamer обрабатывает поток инструкций MCInst и выводит их в требуемом формате с помощью двух подклассов: MCAsmStreamer и MCOBJECTStreamer. Первый преобразует экземпляры MCInst в исходный код на языке ассемблера, а второй – в двоичные инструкции.
- Если генерируется ассемблерный код, вызывается метод MCAsmStreamer::EmitInstruction() и используется архитектурно-зависимый подкласс класса MCInstPrinter, который сохраняет ассемблерные инструкции в файл.
- Если генерируются двоичные инструкции, вызывается специализированная версия MCOBJECTStreamer::EmitInstruction().
- Ассемблер использует специализированный метод MCCodeEmitter::EncodeInstruction(), способный кодиро-

вать и выводить блоки двоичных инструкций в файл архитектурно-зависимым способом.

Инструмент `llc` поддерживает также возможность вывода фрагментов инструкций `MCInst`. Например, чтобы вывести инструкции `MCInst` в комментариях на языке ассемблера, можно воспользоваться следующей командой:

```
$ llc sum.bc -march=x86-64 -show-mc-inst -o -
...
pushq %rbp          ## <MCInst #2114 PUSH64r
                    ## <MCOperand Reg:107>
...
```

Если вы пожелаете вывести в комментариях двоичное представление инструкций, используйте следующую команду:

```
$ llc sum.bc -march=x86-64 -show-mc-encoding -o -
...
pushq %rbp          ## encoding: [0x55]
...
```

Инструмент `llvm-mc` также дает возможность исследовать и использовать инфраструктуру MC. Например, с его помощью можно получить двоичное представление ассемблерной инструкции, для чего достаточно вызвать его с флагом `--show-encoding`. Ниже приводится пример получения двоичного кода для инструкции на языке ассемблера `x86`:

```
$ echo "movq 48879(,%riz), %rax" | llvm-mc -triple=x86_64 --show-encoding
# encoding: [0x48,0x8b,0x04,0x25,0xef,0xbe,0x00,0x00]
```

Данный инструмент реализует также возможность дизассемблирования:

```
$ echo "0x8d 0x4c 0x24 0x04" | llvm-mc --disassemble -triple=x86_64
leal 4(%rsp), %ecx
```

Указав флаг `--show-inst`, можно получить экземпляр `MCInst` для ассемблерной или двоичной инструкции:

```
$ echo "0x8d 0x4c 0x24 0x04" | llvm-mc --disassemble -show-inst
-triple=x86_64
leal 4(%rsp), %ecx          # <MCInst #1105 LEA64_32r
                            # <MCOperand Reg:46>
                            # <MCOperand Reg:115>
                            # <MCOperand Imm:1>
```

```
# <MCOperand Reg: 0>  
# <MCOperand Imm: 4>  
# <MCOperand Reg: 0>>
```

Инфраструктура MC позволяет даже встраивать в LLVM альтернативные инструменты чтения объектных файлов. Например, в настоящее время в LLVM по умолчанию встроены инструменты `llvm-objdump` и `llvm-readobj`. Оба используют библиотеку дисассемблера MC и реализуют те же возможности, что и пакет GNU Binutils (`objdump` и `readelf`).

Реализация собственного прохода для генератора кода

В этом разделе мы покажем, как написать собственный проход для генератора кода, подсчитывающий (непосредственно перед эмиссией кода) число машинных инструкций в каждой функции. В отличие от проходов IR, этот проход нельзя запустить с помощью инструмента `opt` или сообщить о необходимости его выполнения в командной строке. Какие проходы будут выполняться, определяет сам генератор кода. Поэтому мы изменим имеющуюся реализацию генератора так, чтобы она вызывала наш проход. Для этой цели мы решили использовать генератор кода для архитектуры SPARC.

Как рассказывалось в разделе «Демонстрация расширяемого интерфейса проходов» главы 3, «Инструменты и организация», и было показано на рис. 6.1, у нас на выбор есть несколько точек включения своих проходов в процесс. Чтобы воспользоваться такой возможностью, нужно найти подкласс класса `TargetPassConfig`, который реализует выбранный генератор кода. С помощью утилиты `grep` мы выяснили, что требуемая реализация находится в файле `SparcTargetMachine.cpp`:

```
$ cd <llvmsource>/lib/Target/Sparc  
$ vim SparcTargetMachine.cpp # можно использовать любой другой редактор
```

Исследовав класс `SparcPassConfig`, наследующий `TargetPassConfig`, мы обнаружили, что он переопределяет методы `addInstSelector()` и `addPreEmitPass()`, но в родительском классе имеется еще множество методов, которые можно переопределить в потомке, чтобы включить свой проход в другие точки конвейера (см. описание класса по адресу: http://llvm.org/doxygen/classllvm_1_1TargetPassConfig).

html). Нам требуется, чтобы наш проход выполнялся непосредственно перед эмиссией кода, поэтому мы добавим свой код в метод `addPreEmitPass()`:

```
bool SparcPassConfig::addPreEmitPass() {
    addPass(createSparcDelaySlotFillerPass(getSparcTargetMachine()));
    addPass(createMyCustomMachinePass());
}
```

Новая строка выделена жирным – она добавляет наш проход, который создается вызовом функции `createMyCustomMachinePass()`. Однако эта функция пока не определена. Мы создадим новый файл с исходным кодом прохода, где также определим эту функцию. Итак, создайте новый файл с именем `MachineCountPass.cpp` и добавьте в него следующий программный код:

```
#define DEBUG_TYPE "machinecount"
#include "Sparc.h"
#include "llvm/Pass.h"
#include "llvm/CodeGen/MachineBasicBlock.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
class MachineCountPass : public MachineFunctionPass {
public:
    static char ID;
    MachineCountPass() : MachineFunctionPass(ID) {}

    virtual bool runOnMachineFunction(MachineFunction &MF) {
        unsigned num_instr = 0;
        for (MachineFunction::const_iterator I = MF.begin(), E = MF.end();
             I != E; ++I) {
            for (MachineBasicBlock::const_iterator BBI = I->begin(),
                 BBE = I->end(); BBI != BBE; ++BBI) {
                ++num_instr;
            }
        }
        errs() << "mcount --- " << MF.getName() << " has "
                << num_instr << " instructions.\n";
        return false;
    }
};

FunctionPass *llvm::createMyCustomMachinePass() {
    return new MachineCountPass();
}
```

```
}  
  
char MachineCountPass::ID = 0;  
static RegisterPass<MachineCountPass> X("machinecount",  
                                         "Machine Count Pass");
```

В первой строке определяется макрос `DEBUG_TYPE`, чтобы потом иметь возможность отлаживать проход с использованием флага `-debug-only=machinecount`; однако в данном примере вывод отладочной информации не реализован. Остальной код очень похож на тот, что мы писали в предыдущей главе, когда реализовали проход IR. Различия заключаются в следующем:

- В числе заголовочных файлов подключаются `MachineBasicBlock.h`, `MachineFunction.h` и `MachineFunctionPass.h` с определениями классов, которые используются для извлечения информации о `MachineFunction` и позволяют подсчитать число инструкций. Также подключается заголовочный файл `Sparc.h`, потому что туда мы добавим объявление функции `createMyCustomMachinePass()`.
- Далее определяется класс, наследующий `MachineFunctionPass` вместо `FunctionPass`.
- Переопределяется метод `runOnMachineFunction()` вместо `runOnFunction()`. Кроме того, реализация метода существенно отличается от реализации `runOnFunction()`. Метод выполняет итерации по всем экземплярам `MachineBasicBlock` в текущем экземпляре `MachineFunction`. В каждом экземпляре `MachineBasicBlock` подсчитывается число инструкций, с применением идиомы `begin()/end()`.
- Определяется функция `createMyCustomMachinePass()`, создающая данный проход и добавляющая его перед эмиссией кода.

После определения функции `createMyCustomMachinePass()` ее следует объявить в заголовочном файле. Для этого отредактируйте заголовочный файл `Sparc.h`, как показано далее. Добавьте свое объявление после объявления `createSparcDelaySlotFillerPass()`:

```
FunctionPass *createSparcISelDag(SparcTargetMachine &TM);  
FunctionPass *createSparcDelaySlotFillerPass(TargetMachine &TM);  
FunctionPass *createMyCustomMachinePass();
```

Теперь пришло время собрать обновленный генератор кода для архитектуры SPARC. Если до сих пор вы не имели возможности за-

няться настройкой сборки LLVM, обращайтесь к главе 1, «Сборка и установка LLVM». Если у вас уже имеется каталог сборки с настроенной конфигурацией проекта, перейдите в этот каталог и выполните команду `make`, чтобы скомпилировать обновленный генератор кода. После этого вы можете вновь установить LLVM с измененным генератором кода для архитектуры SPARC или, если хотите, можете просто вызывать вновь скомпилированный инструмент `llc` непосредственно из каталога сборки, не вызывая команду `make install`:

```
$ cd <llvm-build>
$ make
$ Debug+Asserts/bin/llc -march=sparc sum.bc
```

```
mcount --- sum has 8 instructions.
```

Желающие узнать, какое место в конвейере занимает новый проход, могут выполнить команду:

```
$ Debug+Asserts/bin/llc -march=sparc sum.bc -debug-pass=Structure
(...)
```

```
Branch Probability Basic Block Placement
SPARC Delay Slot Filler
Machine Count Pass
MachineDominator Tree Construction
Machine Natural Loop Construction
Sparc Assembly Printer
```

```
mcount --- sum has 8 instructions.
```

Как видите, наш проход был вставлен сразу после прохода SPARC Delay Slot Filler и перед проходом Sparc Assembly Printer.

В заключение

В этой главе мы представили общий обзор особенностей генераторов кода в LLVM. Мы познакомились с разными этапами работы генератора и внутренними представлениями инструкций, которые изменяются в ходе компиляции. Мы обсудили этапы выбора инструкций, планирования, распределения регистров, эмиссии кода и приемы использования инструментов, используя которые вы сможете поэкспериментировать с этими этапами. Прочитав эту главу до конца, вы теперь знаете, как читать вывод команды `llc -debug` с подробностями, описывающими работу генератора кода, и имеете более или менее полное представление о его внутреннем устройстве. Если у вас появится желание создать собственный генератор кода, обращайтесь

к официальному руководству по адресу: <http://llvm.org/docs/WritingAnLLVMBackend.html>. Желаящим поближе познакомиться с устройством генераторов мы рекомендуем обратиться к статье <http://llvm.org/docs/CodeGenerator.html>.

В следующей главе мы познакомимся с инфраструктурой динамической (Just-in-Time) компиляции в LLVM, которая дает возможность генерировать выполняемый код по мере необходимости.

ГЛАВА 7.

Динамический компилятор

Динамический (Just-in-Time, JIT) компилятор LLVM – это механизм динамической трансляции на уровне функций. Чтобы понять, что есть динамический (JIT) компилятор, вернемся к истокам происхождения этого термина.¹ Этот термин родился в промышленном производстве и обозначает концепцию, основная идея которой заключается в следующем: если производственное расписание задано, то можно так организовать движение материальных потоков, что все материалы, компоненты и полуфабрикаты будут поступать в необходимом количестве, в нужное место и точно к назначенному сроку для производства, сборки или реализации готовой продукции, благодаря чему отпадает необходимость хранить запасы ресурсов. Эта аналогия отлично ложится на компиляцию, потому что JIT-компилятор не сохраняет двоичный код на диск (запасы) и компилирует программу по частям, когда в них возникает необходимость, прямо во время выполнения программы. Несмотря на широкое распространение этого термина, вы можете встретить и другие названия, такие как отложенная или «ленивая» компиляция.

Преимущество стратегии динамической компиляции заключается в обладании полной информации о целевой архитектуре, на которой выполняется программа. Это дает JIT-системе возможность оптимизировать код под конкретный процессор. Кроме того, иногда программы доступны во время выполнения только в исходном коде и нет никакой возможности скомпилировать их заранее. Например, драйвер GPU компилирует шейдерные программы прямо во время выполнения, то же происходит в браузерах со сценариями на JavaScript. В этой главе мы исследуем JIT-систему в LLVM и затронем следующие темы:

- ♦ класс `llvm::JIT` и его инфраструктура;
- ♦ порядок использования класса `llvm::JIT` для динамической компиляции;

¹ Дословно фраза «just in time» переводится как «точно в срок». – Прим. перев.

- ♦ порядок использования `GenericValue` для упрощения вызовов функций;
- ♦ класс `llvm::MCJIT` и его инфраструктура;
- ♦ порядок использования класса `llvm::MCJIT` для динамической компиляции

Основы механизма динамической компиляции в LLVM

Динамический (JIT) компилятор действует на уровне функций, потому что может компилировать только по одной функции за один раз. Это определяет уровень детализации работы компилятора, который является важным параметром для организации JIT-системы. Компилируя функции по требованию, система работает только с функциями, которые действительно вызываются программой. Например, если программа имеет несколько функций, но при запуске ей были переданы недопустимые аргументы командной строки, JIT-система уровня функций скомпилирует только функцию, которая выводит справочное сообщение, а не всю программу.

Примечание. Теоретически можно было бы увеличить уровень детализации еще больше и компилировать только те маршруты внутри функции, которые действительно выполняются. Для этого JIT-система обладает всем необходимым: сведениями о том, по какому пути пойдет выполнение программы при тех или иных исходных данных. Однако, JIT-система в LLVM не поддерживает такую выборочную компиляцию. Динамическая компиляция все еще остается предметом бесконечных дискуссий, опирается на массу компромиссных решений, достойных тщательного изучения, и не так-то просто сказать, какая стратегия дает лучшие результаты. В настоящее время сообществом исследователей накоплен более чем 20-летний опыт в области динамической компиляции и, тем не менее, каждый год появляются все новые труды, авторы которых пытаются решить открытые вопросы.

Работа JIT-системы заключается в том, чтобы компилировать и тут же выполнять функции LLVM IR. Для компиляции JIT-система использует генератор кода LLVM, создающий блок двоичных инструкций для данной архитектуры и возвращающий указатель на скомпилированную функцию, которая может быть немедленно выполнена.

Совет. Желющие могут прочитать интересную статью, где сравниваются открытые решения по реализации JIT-компиляторов: <http://eli.thegreenplace.net/2014/01/15/some-thoughts-on-llvm-vs-libjit>. В статье анализируются LLVM и libjit, небольшой открытый проект, целью которого является реализация JIT-компилятора. Проект LLVM более известен как статический компилятор, чем как JIT-система, потому что для динамической компиляции огромное значение играет время, затрачиваемое каждым проходом, которое прибавляется к времени фактического выполнения скомпилированного кода. В инфраструктуре LLVM больше внимания уделяется поддержке медленной компиляции с широкими возможностями оптимизации кода, сопоставимыми с GCC, чем быстрой компиляции с посредственной оптимизацией, которая могла бы конкурировать с другими JIT-системами. Тем не менее, проект LLVM с успехом используется в JIT-системе механизма Webkit JavaScript, где он выбран в качестве основы для компонента **Fourth Tier LLVM (FTL)** (см. <http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html>). Так как этот компонент используется только для компиляции JavaScript-приложений с большим временем выполнения, агрессивные оптимизации LLVM оказываются способны помочь в увеличении производительности, даже при том, что они имеют немалые накладные расходы. Объяснить это несложно: если приложение работает достаточно долго, накладные расходы на дорогостоящие оптимизации компенсируются за счет многократного выполнения оптимизированного кода. Подробнее об этом компромиссе можно прочитать в статье «Modeling Virtual Machines Misprediction Overhead», написанной Дивином Цесаром (Divino César) с соавторами и опубликованной в материалах симпозиума IISWC 2013. В этой статье исследуются проблемы увеличения затрат на JIT-компиляцию из-за неправильно выбранных параметров оптимизации для кода, который этого не заслуживает. Эта проблема проявляется, когда JIT-система тратит слишком много времени на оптимизацию фрагмента, который выполняется всего один-два раза.

Введение в механизм выполнения

JIT-система в LLVM использует механизм выполнения для поддержки модулей LLVM. В файле `<llvm_source>/include/llvm/ExecutionEngine/ExecutionEngine.h` объявлен класс `ExecutionEngine`, созданный для поддержки выполнения в JIT-системе или в интерпретаторе (см. примечание ниже). В общем случае механизм выполнения отвечает за управление выполнением всей гостевой программы, анализ следующего фрагмента программы, подлежащего выполнению, и принятия надлежащих мер для его выполнения. Когда выполняется динамическая компиляция, совершенно необходимо иметь диспетче-

ра выполнения, который бы управлял компиляцией и выполнял гостевую программу (фрагментами). Однако, в случае с классом `ExecutionEngine`, часть обязанностей по выполнению возлагается на вас, то есть клиента. Класс `ExecutionEngine` может выполнить компиляцию и сохранить выполняемый код в памяти, но оставляет за вами решение о том, следует ли выполнять этот код.

Помимо выполнения модулей LLVM, механизм поддерживает также следующие возможности:

- **ленивая, или отложенная компиляция (lazy compilation):** когда компиляция функций выполняется только в случае их вызова. Когда ленивая компиляция выключена, механизм компилирует функции при первых попытках получить указатели на них;
- **компиляция внешних глобальных переменных:** включает разрешение символов и распределение памяти для сущностей за пределами текущего модуля LLVM;
- **поиск и разрешение внешних символов через `dlsym`:** тот же самый процесс, что используется для загрузки динамических разделяемых объектов (**Dynamic Shared Object, DSO**).

В LLVM имеется две реализации механизма выполнения JIT: классы `llvm::JIT` и `llvm::MCJIT`. Объект `ExecutionEngine` создается вызовом метода `ExecutionEngine::EngineBuilder()`, которому в виде аргумента передается модуль `IRModule`. Затем, вызовом `ExecutionEngine::create()` создается экземпляр JIT или MCJIT механизма выполнения. Эти две реализации имеют существенные отличия, о которых постоянно будет упоминаться в этой главе.

Примечание. Интерпретаторы реализуют альтернативную стратегию выполнения гостевого кода, то есть код, который не поддерживается аппаратной платформой (или хост-платформой) непосредственно. Например, промежуточное представление LLVM IR является гостевым кодом для платформы x86, потому что процессоры x86 не могут выполнять код LLVM IR непосредственно. В отличие от JIT-компиляторов, интерпретаторы читают отдельные инструкции, декодируют их и выполняют, имитируя функциональные возможности физического процессора программным способом. Даже при том, что интерпретаторы не тратят время на запуск компилятора для трансляции гостевого кода, они, как правило, действуют значительно медленнее, за исключением случаев, когда накладные расходы на компиляцию гостевого кода оказываются выше накладных расходов на интерпретацию.

Управление памятью

В общем случае работа механизма JIT заключается в сохранении блоков двоичного кода в памяти, созданных классом `ExecutionManager`. Впоследствии программа сможет выполнить этот код, просто вызвав его как функцию по указателю, который возвращает `ExecutionManager`. В данном контексте управление памятью заключается в выполнении рутинных операций, таких как выделение, освобождение памяти для загружаемых библиотек и поддержка системы разрешений.

Оба класса, JIT и MSJIT, реализуют собственный класс управления памятью, наследующий базовый класс `RTDyldMemoryManager`. Любой клиент `ExecutionEngine` также может реализовать собственный подкласс класса `RTDyldMemoryManager` и с его помощью определять, где должны храниться разные JIT-компоненты. Объявление интерфейса находится в файле `<llvm_source>/include/llvm/ExecutionEngine/RTDyldMemoryManager.h`.

Например, класс `RTDyldMemoryManager` объявляет следующие методы:

- `allocateCodeSection()` и `allocateDataSection()`: выделяют память для хранения выполняемого кода и данных. Клиент управления памятью может следить за выделением сегментов с помощью внутреннего идентификатора сегмента, передаваемого в виде аргумента.
- `getSymbolAddress()`: возвращает адрес символа, доступно в библиотеке. Имейте в виду, что этот метод не может использоваться для получения адресов символов, сгенерированных в ходе JIT-компиляции. Метод принимает экземпляр `std::string` с именем искомого символа.
- `finalizeMemory()`: должен вызываться сразу после загрузки объекта, когда появляется возможность установить разрешения на доступ к памяти. Например, сгенерированный код не получится запустить до вызова этого метода. Как описывается далее в этой главе, этот метод в большей степени предназначен для клиентов MSJIT, чем для клиентов JIT.

Клиенты могут предоставлять свои реализации управления памятью – классы `JITMemoryManager` и `SectionMemoryManager` для механизмов JIT и MSJIT, соответственно.



Введение в инфраструктуру `llvm::JIT`

Класс `JIT` и его инфраструктура – это достаточно старый механизм, реализованный с применением разных компонентов генератора кода LLVM. Он будет удален из проекта после выхода версии LLVM 3.5. Этот механизм в значительной степени является архитектурно-независимым и тем не менее для каждой архитектуры должен быть реализован этап эмиссии двоичного кода.

Запись блоков двоичного кода в память

Класс `JIT` выводит двоичные инструкции с помощью класса `JITCodeEmitter`, наследующего базовый класс `MachineCodeEmitter`. Класс `MachineCodeEmitter` используется для эмиссии машинного кода и не связан с новой инфраструктурой **Machine Code (MC)** – несмотря на почтенный возраст, он все еще присутствует для поддержки класса `JIT`. Одним из недостатков этого класса является ограниченный круг поддерживаемых архитектур, а для поддерживаемых архитектур доступны не все их особенности.

Класс `MachineCodeEmitter` имеет методы, упрощающие решение следующих задач:

- распределение памяти (`allocateSpace()`) для текущей генерируемой функции;
- запись блоков двоичного кода в память (`emitByte()`, `emitWordLE()`, `emitWordBE()`, `emitAlignment()` и другие);
- слежение за текущим адресом в буфере (указатель на адрес, куда будет записана следующая инструкция);
- добавление смещений относительно адресов инструкций в буфере.

Задача записи байтов в память выполняется еще одним классом, `JITCodeEmitter`, вовлеченным в процесс эмиссии кода. Именно подкласс `JITCodeEmitter` реализует специфическую функциональность и средства управления `JIT`-компиляцией. Класс `JITCodeEmitter` весьма прост и просто записывает байты в буфера, но есть еще один класс, `JITEmitter`, имеющий следующие улучшения:

- специализированный диспетчер памяти, `JITMemoryManager`, упоминавшийся выше (а также описываемый в следующем разделе);

- механизм разрешения символов (`JITResolver`) для слежения и разрешения вызовов функций, которые еще не были скомпилированы. Является основой для отложенной компиляции функций.

JITMemoryManager

Класс `JITMemoryManager` (см. `<llvm_source>/include/llvm/ExecutionEngine/JITMemoryManager.h`) реализует низкоуровневое управление памятью и выделяет буферы для использования классами, упоминавшимися выше. Помимо методов, унаследованных от `RTDyldMemoryManager`, предоставляет собственные методы, такие как `allocateGlobal()` (выделяет память для единственной глобальной переменной) и `startFunctionBody()` (вызывается классом `JIT`, когда ему необходимо выделить память с соответствующими разрешениями для сохранения выполняемых инструкций).

Внутренне класс `JITMemoryManager` использует диспетчера памяти `JITSlabAllocator` (`<llvm_source>/lib/ExecutionEngine/JIT/JITMemoryManager.cpp`) и блоки памяти `MemoryBlock` (`<llvm_source>/include/llvm/Support/Memory.h`).

Механизмы вывода целевого кода

Для каждой целевой архитектуры реализуется проход эмиссии машинного кода `<Целевая_архитектура>CodeEmitter` (см. `<llvm_source>/lib/Target/<Целевая_архитектура>/<Целевая_архитектура>CodeEmitter.cpp`), который кодирует инструкции в блоках и с помощью `JITCodeEmitter` записывает их в память. Класс `MipsCodeEmitter`, например, выполняет обход всех базовых блоков функции и для каждой машинной инструкции (МИ) вызывает `emitInstruction()`:

```
(...)
MCE.startFunction(MF);

for (MachineFunction::iterator MBB = MF.begin(), E = MF.end();
     MBB != E; ++MBB){
MCE.StartMachineBasicBlock(MBB);
for (MachineBasicBlock::instr_iterator I = MBB->instr_begin(),
     E = MBB->instr_end(); I != E;){
    emitInstruction(*I++, *MBB);
}
(...)

```

Все команды архитектуры MIPS32 имеют фиксированную длину 4 байта, что делает реализацию `emitInstruction()` чрезвычайно простой:

```
void MipsCodeEmitter::emitInstruction(
    MachineBasicBlock::instr_iterator MI,
    MachineBasicBlock &MBB) {
    ...
    MCE.processDebugLoc(MI->getDebugLoc(), true);
    emitWord(getBinaryCodeForInstr(*MI));
    ++NumEmitted; // Запомнить число записанных инструкций
    ...
}
```

Метод `emitWord()` является оберткой вокруг `JITCodeEmitter`, а метод `getBinaryCodeForInstr()` генерируется для каждой архитектуры на основе описания кодирования инструкций на языке `TableGen`. Класс `<Целевая_архитектура>CodeEmitter` должен также реализовать собственные методы для кодирования операндов и других архитектурно-зависимых элементов. Например, в архитектуре MIPS для кодирования операнда `mem` необходимо использовать метод `getMemEncoding()` (см. `<llvm_source>/lib/Target/Mips/MipsInstrInfo.td`):

```
def mem : Operand<iPTR> {
    (...)
    let MIOperandInfo = (ops ptr_rc, simml6);
    let EncoderMethod = "getMemEncoding";
    (...)
}
```

Соответственно, `MipsCodeEmitter` должен реализовать метод `MipsCodeEmitter::getMemEncoding()`, соответствующий этому описанию на языке `TableGen`. На рис. 7.1 показаны отношения между некоторыми механизмами эмиссии кода и инфраструктурой JIT:

Информация о целевой архитектуре

Для поддержки динамической компиляции каждая архитектура должна предоставлять класс, наследующий `TargetJITInfo` (см. `include/llvm/Target/TargetJITInfo.h`), такой как `MipsJITInfo` или `x86JITInfo`. Класс `TargetJITInfo` определяет интерфейс доступа к функциональности JIT и должен быть реализован для каждой архитектуры. Далее мы представим группу примеров использования такой функциональности.

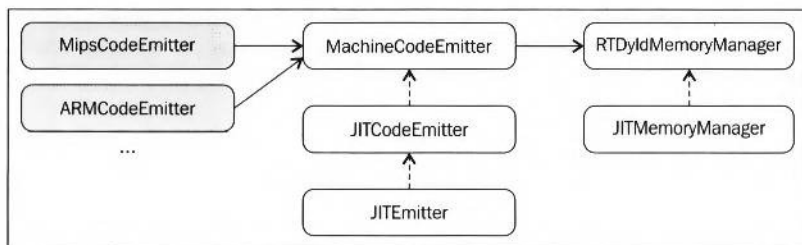


Рис. 7.1. Отношения между механизмами эмиссии кода и инфраструктурой JIT

- Для поддержки ситуаций, когда механизму выполнения потребуется повторно скомпилировать функцию – например, в случае изменения исходного кода функции, – для каждой целевой архитектуры должен быть реализован метод `TargetJITInfo::replaceMachineCodeForFunction()`, изменяющий старый адрес функции на новый в инструкциях перехода или вызова. Это необходимо для самомодифицирующегося кода.
- Метод `TargetJITInfo::relocate()` должен изменять все ссылки на символы в текущей функции так, чтобы они ссылались на правильные адреса в памяти, подобно тому, как действует динамический компоновщик.
- Метод `TargetJITInfo::emitFunctionStub()` должен генерировать «заглушку» – функцию, вызывающую другую функцию по указанному адресу. Для каждой архитектуры также должно быть реализовано свойство `TargetJITInfo::StubLayout`, возвращающее информацию о размере в байтах и выравнивании заглушки. Эта информация используется экземплярами `JITEmitter` для выделения памяти под новую заглушку перед ее созданием.

Цель методов класса `TargetJITInfo` состоит не в том, чтобы генерировать обычные инструкции, как в теле функции, но они должны возвращать специальные инструкции для создания заглушки и изменять адреса вызовов и переходов. Когда инфраструктура JIT появилась на свет, в ней отсутствовал интерфейс, упрощающий создание автономных инструкций, которые можно было бы использовать за пределами `MachineBasicBlock`. Именно такой интерфейс реализует `MCInsts` для `MCJIT`. Без `MCInsts` старая инфраструктура JIT вынуждала вручную кодировать инструкции.

Чтобы показать, как реализация `<Target>JITInfo` должна была вручную кодировать инструкции, мы решили привести фрагмент реализации `MipsJITInfo::emitFunctionStub()` (см. `<llvm_source>/lib/Target/Mips/MipsJITInfo.cpp`), который генерирует четыре инструкции:

```
...
// lui $t9, %hi(EmittedAddr)
// addiu $t9, $t9, %lo(EmittedAddr)
// jalr $t8, $t9
// nop
if (IsLittleEndian) {
    JCE.emitWordLE(0xf << 26 | 25 << 16 | Hi);
    JCE.emitWordLE(9 << 26 | 25 << 21 | 25 << 16 | Lo);
    JCE.emitWordLE(25 << 21 | 24 << 11 | 9);
    JCE.emitWordLE(0);
}
...
```

Практика применения класса JIT

Класс `JIT` является наследником класса `ExecutionEngine` и объявлен в файле `<llvm_source>/lib/ExecutionEngine/JIT/JIT.h`. Класс `JIT` – это точка входа в механизм компиляции функций, основанный на инфраструктуре `JIT`.

Метод `ExecutionEngine::create()` вызывает `JIT::createJIT()`, с диспетчером по умолчанию `JITMemoryManager`. Далее конструктор `JIT` выполняет следующие операции:

- создает экземпляр `JITEmitter`;
- инициализирует объект с информацией о целевой архитектуре;
- добавляет проходы в конвейер генератора кода;
- добавляет проход `<Целевая_архитектура>CodeEmitter` для запуска в конце.

Механизм хранит объект `PassManager` для вызова всех проходов генератора кода и `JIT` и получения выполняемого кода, когда запрашивается `JIT`-компиляция функции.

Все действия механизма `JIT`-компиляции мы проиллюстрируем на примере функции в файле `sum.bc` с биткодом, использовавшимся в главе 5, «Промежуточное представление LLVM», и в главе 6, «Генератор выполняемого кода». Наша цель в том, чтобы скомпилировать функцию `Sum` и с помощью `JIT`-системы вычислить две суммы с разными аргументами, поставляемыми во время выполнения. Для этого нужно выполнить следующие шаги:

1. Создать новый файл `sum-jit.cpp` и подключить ресурсы механизма JIT выполнения:

```
#include "llvm/ExecutionEngine/JIT.h"
```

2. Подключить другие заголовочные файлы, где определяются инструменты для чтения и записи биткода, интерфейс к контексту, и импортировать пространство имен LLVM:

```
#include "llvm/ADT/OwningPtr.h"
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/system_error.h"
#include "llvm/Support/TargetSelect.h"
using namespace llvm;
```

Метод `InitializeNativeTarget()` устанавливает в качестве целевой архитектуру хоста и гарантирует загрузку целевых библиотек. Как обычно, нам потребуется потоко-безопасный объект контекста `LLVMContext` и объект `MemoryBuffer` для чтения биткода из файла на диске:

```
int main() {
    InitializeNativeTarget();
    LLVMContext Context;
    std::string ErrorMessage;
    OwningPtr<MemoryBuffer> Buffer;
```

3. Прочитать биткод из файла на диске с помощью метода `getFile()`, как показано ниже:

```
if (MemoryBuffer::getFile("./sum.bc", Buffer)) {
    errs() << "sum.bc not found\n";
    return -1;
}
```

4. Функция `ParseBitcodeFile` читает данные из `MemoryBuffer` и генерирует соответствующий класс модуля LLVM `Module`:

```
Module *M = ParseBitcodeFile(Buffer.get(), Context,
                             &ErrorMessage);

if (!M) {
    errs() << ErrorMessage << "\n";
    return -1;
}
```

5. Создать экземпляр `ExecutionEngine` с помощью фабрики `EngineBuilder` и ее метода `create`:

```
OwningPtr<ExecutionEngine> EE(EngineBuilder(M).create());
```

Этот метод по умолчанию создает и настраивает механизм выполнения JIT; он вызывает конструктор JIT, который создаст объекты `JITemitter` и `PassManager`, и инициализирует все проходы генератора кода. В этой точке, даже при том, что механизм имеет в своем распоряжении модуль `LLVM Module`, пока еще никакой код не скомпилирован.

Чтобы скомпилировать функцию, нужно вызвать метод `getPointerToFunction()`, возвращающий указатель на скомпилированную функцию. Если функция еще не была скомпилирована, метод компилирует ее и возвращает указатель на только что скомпилированную функцию. На рис. 7.2 представлена диаграмма процесса компиляции.

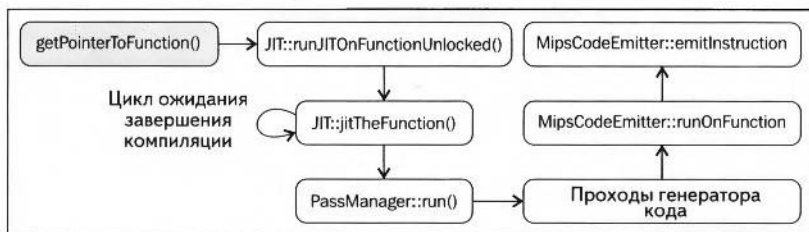


Рис. 7.2. Процесс JIT-компиляции функции

6. Извлечь объект `Function` с промежуточным представлением функции `sum` вызовом метода `getFunction()`:

```
Function *SumFn = M->getFunction("sum");
```

В этой точке выполняется JIT-компиляция:

```
int (*Sum)(int, int) = (int (*)(int, int))
EE->getPointerToFunction(SumFn);
```

Далее нужно выполнить приведение типа указателя, чтобы он соответствовал сигнатуре функции. Для функции `Sum` определен LLVM-прототип `define i32 @sum(i32 %a, i32 %b)`, которому соответствует C-прототип `int (*)(int, int)`.

Как вариант, вместо метода `getPointerToFunction()`, выполняющего компиляцию немедленно, можно использовать метод `getPointerToFunctionOrStub()`, реализующий отло-

женную компиляцию. Этот метод генерирует функцию-заглушку и возвращает указатель на нее, если целевая функция еще не была скомпилирована и разрешена отложенная компиляция. Заглушка – это небольшая функция, которая содержит инструкцию, которая позднее будет замещена инструкцией перехода/вызова фактической функции.

7. Далее нужно вызвать скомпилированную функцию `Sum`, на которую ссылается указатель `Sum`:

```
int res = Sum(4,5);
outs() << "Sum result: " << res << "\n";
```

В режиме отложенной компиляции будет вызвана функция-заглушка, которая вызовет JIT-компилятор и скомпилирует фактическую функцию. Затем заглушка исправит адрес вызова, подставив в него адрес фактической функции. Если только исходная функция `Sum` в модуле `Module` не изменится, она никогда не будет компилироваться повторно.

8. Вызвать `Sum` еще раз, чтобы вычислить следующую сумму:

```
res = Sum(res, 6);
outs() << "Sum result: " << res << "\n";
```

В режиме отложенной компиляции, поскольку исходная функция уже была скомпилирована при первом вызове, второй вызов выполнит скомпилированную версию функции.

9. Мы благополучно вычислили две суммы с помощью функции `Sum`, скомпилированной JIT-компилятором. Теперь, вызовом `llvm_shutdown()`, можно освободить память, занятую механизмом выполнения, и вернуть управление:

```
EE->freeMachineCodeForFunction(SumFn);
llvm_shutdown();
return 0;
}
```

Чтобы скомпилировать и скомпоновать `sum-jit.cpp`, можно воспользоваться следующей командой:

```
$ clang++ sum-jit.cpp -g -O3 -rdynamic -fno-rtti $(llvm-config
--cppflags --ldflags --libs jit native irreader) -o sum-jit
```

В качестве альтернативы можно написать `Makefile`, как описывается в главе 3, «Инструменты и организация», добавив флаг `-rdynamic` и изменив вызов `llvm-config` так, чтобы задействовать библиотеки, перечисленные в команде выше. Флаг `-rdynamic` гарантирует разре-

шение имен внешних функций во время выполнения, однако в этом примере внешние функции не используются.

Запустите скомпилированный выполняемый файл и проверьте результат:

```
$ ./sum-jit  
  
Sum result: 9  
Sum result: 15
```

Обобщенные значения

В предыдущем примере мы приводили указатель на функцию к типу ее прототипа, чтобы иметь возможность использовать привычный способ вызова функции в стиле языка C. Однако, при наличии множества функций с разными сигнатурами, необходим более гибкий способ вызова функций.

Механизм выполнения предоставляет еще один способ вызова динамически скомпилированных функций. Метод `runFunction()` компилирует и вызывает функцию с аргументами, значения которых передаются ему в виде вектора `GenericValue` – он не требует предварительно вызывать `getPointerToFunction()`.

Структура `GenericValue` определена в `<llvm_source>/include/llvm/ExecutionEngine/GenericValue.h` и может хранить значение любого распространенного типа. Давайте изменим наш последний пример, задействовав в нем `runFunction()` вместо `getPointerToFunction()` и операции приведения типа.

Сначала создайте файл `sum-jit-gv.cpp`, где будет храниться исходный код новой версии, и подключите в нем заголовочный файл `GenericValue.h`:

```
#include "llvm/ExecutionEngine/GenericValue.h"
```

Скопируйте остальной код из файла `sum-jit.cpp`. А теперь сосредоточимся на изменениях. После инициализации указателя `SumFn` `Function` создайте `FnArgs` – вектор типа `GenericValue` и заполните его целочисленными значениями с помощью интерфейса `APInt` (`<llvm_source>/include/llvm/ADT/APInt.h`). Используйте два 32-разрядных целых числа, чтобы соответствовать прототипу `sum(i32 %a, i32 %b)`:

```
(...)  
Function *SumFn = M->getFunction("sum");  
std::vector<GenericValue> FnArgs(2);  
FnArgs[0].IntVal = APInt(32, 4);  
FnArgs[1].IntVal = APInt(32, 5);
```

Вызовите метод `runFunction()`, передав ему указатель на функцию и вектор с аргументами. Он скомпилирует функцию и вызовет ее. Результат метода также имеет тип `GenericValue` и должен использоваться соответственно:

```
GenericValue Res = EE->runFunction(SumFn, FnArgs);
outs() << "Sum result: " << Res.IntVal << "\n";
```

Повторите ту же процедуру, чтобы получить вторую сумму:

```
FnArgs[0].IntVal = Res.IntVal;
FnArgs[1].IntVal = APInt(32, 6);
Res = EE->runFunction(SumFn, FnArgs);
outs() << "Sum result: " << Res.IntVal << "\n";
(...)
```

Введение в инфраструктуру llvm::MCJIT

Класс `MCJIT` является новейшей реализацией механизма динамической компиляции для LLVM. От прежней реализации он отличается поддержкой инфраструктуры MC, о которой рассказывалось в главе 6, «Генератор выполняемого кода». Инфраструктура MC обеспечивает однородное представление инструкций и совместно используется ассемблером, дизассемблером, инструментом вывода сборок и классом `MCJIT`.

Первое преимущество от использования библиотеки MC заключается в централизованном хранении информации о целевой архитектуре – достаточно определить кодировки инструкций один раз и они будут использоваться всеми подсистемами. Соответственно, при разработке генератора кода для своей архитектуры вы автоматически получаете поддержку JIT-функциональности.

Инфраструктура `llvm::JIT` будет удалена из проекта после выхода версии LLVM 3.5 и ее заменит инфраструктура `llvm::MCJIT`. Возникает законный вопрос: «Зачем тогда мы тратили время на исследование старой инфраструктуры JIT?». Ответ прост. Да, это разные реализации, но они обе используют универсальный класс `ExecutionEngine`, поэтому большинство понятий применимы к обоим инфраструктурам. Но самое важное – инфраструктура `MCJIT` в версии LLVM 3.4 не поддерживает некоторые возможности, такие как отложенная компиляция, и пока не готова заменить старую инфраструктуру `JIT`.

Механизм MCJIT

Экземпляр механизма MCJIT создается точно так же, как экземпляр механизма JIT, вызовом `ExecutionEngine::create()`. Этот метод вызовет `MCJIT::createJIT()`, который в свою очередь выполнит конструктор MCJIT. Класс MCJIT объявлен в файле `<llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.h`. Метод `createJIT()` и конструктор MCJIT реализованы в `<llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp`.

Конструктор MCJIT создает экземпляр `SectionMemoryManager`; добавляет модуль LLVM в его внутренний контейнер, `OwningModuleContainer`, и инициализирует информацию о целевой архитектуре.

Состояния модуля

Класс MCJIT назначает разные состояния экземплярам `Module` модулей LLVM в процессе работы, которые представляют разные стадии компиляции модуля.

- **Добавлен:** модули в этом состоянии содержат множество модулей, еще не скомпилированных, но уже добавленных в механизм выполнения. Это состояние позволяет модулям экспортировать определения функций для других модулей и задерживать их компиляцию до нужного момента.
- **Загружен:** модули в этом состоянии уже скомпилированы, но пока не готовы к выполнению. Смещения к адресам еще не применялись и страницам памяти еще не давались соответствующие разрешения. Клиенты, желающие перераспределить скомпилированные функции в памяти, могут избежать повторной их компиляции, используя модули в состоянии «загружен».
- **Готов:** модули в этом состоянии содержат функции, готовые к выполнению. Функции в таких модулях не могут перераспределяться, потому что все смещения уже применены.

Одно из важнейших отличий между JIT и MCJIT заключается в состояниях модулей. В MCJIT модуль должен быть готов, прежде чем к нему можно будет обратиться за получением адресов символов (функций и других глобальных сущностей).

Метод `MCJIT::finalizeObject()` преобразует добавленные модули в загруженные и затем в готовые. Сначала он генерирует загруженные модули вызовом `generateCodeForModule()`. После этого все

модули переводятся в готовое состояние вызовом метода `finalizeLoadedModules()`.

В отличие от метода `getPointerToFunction()` в инфраструктуре JIT, метод `MCJIT::getPointerToFunction()` требует передачи ему объекта `Module` в готовом состоянии. То есть, перед вызовом метода `MCJIT::getPointerToFunction()` обязательно должен быть вызван метод `MCJIT::finalizeObject()`.

В LLVM 3.4 появился новый метод `getFunctionAddress()`, устраняющий это ограничение, а метод `getPointerToFunction()` объявлен нерекондуемым к использованию при работе с MCJIT. Этот новый метод загружает модули и подготавливает их к выполнению, благодаря чему отпадает необходимость явно вызывать `finalizeObject()`.

Примечание. Обратите внимание, что в старой инфраструктуре JIT функции компилируются отдельно и выполняются с помощью механизма выполнения. В инфраструктуре MCJIT, напротив, прежде чем можно будет вызвать какую-либо функцию, должен быть скомпилирован весь модуль целиком (все функции). Из-за такого уменьшения избирательности можно сказать, что единицей компиляции в этой инфраструктуре является уже не функция, а модуль.

Как MCJIT компилирует модули

Процедура компиляции устанавливает состояние «загружен» в объекте `Module` и запускается вызовом `MCJIT::generateCodeForModule()` (см. `<llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp`). Этот метод выполняет следующие операции.

- Создает экземпляр `ObjectBuffer` для хранения объекта `Module`. Если объект `Module` уже находится в состоянии «загружен» (то есть, скомпилирован), для его извлечения используется интерфейс `ObjectCache`, чтобы избежать повторной компиляции.
- Исходя из предположения, что никакого кэша с кодом не существует, вызовом `MCJIT::emitObject()` выполняется эмиссия кода. Результат сохраняется в объекте `ObjectBufferStream` (наследует класс `ObjectBuffer` и добавляет поддержку потоков данных).
- Динамический компоновщик `RuntimeDyld` загружает получившийся объект `ObjectBuffer` и конструирует таблицу сим-

волов вызовом `RuntimeDyld::loadObject()`. Этот метод возвращает объект `ObjectImage`.

- Модулю присваивается состояние «загружен».

Объект буфера, кэш и образ

Класс `ObjectBuffer` (`<llvm_source>/include/llvm/ExecutionEngine/ObjectBuffer.h`) реализует обертку вокруг класса `MemoryBuffer` (`<llvm_source>/include/llvm/Support/MemoryBuffer.h`).

Класс `MemoryBuffer` используется подклассами класса `MCObjectStreamer` для эмиссии инструкций и данных в память. Кроме того, класс `ObjectCache` напрямую обращается к экземплярам `MemoryBuffer` и способен извлекать из них экземпляры `ObjectBuffer`.

Класс `ObjectBufferStream` наследует `ObjectBuffer` и расширяет его реализацией поддержки стандартных потоковых операторов C++ (таких как `>>` и `<<`) и упрощает выполнение операций чтения/записи из памяти буфера.

Объект `ObjectImage` (`<llvm_source>/include/llvm/ExecutionEngine/ObjectImage.h`) используется для хранения загруженных модулей и имеет прямой доступ к экземплярам `ObjectBuffer` и `ObjectFile`. Объект `ObjectFile` (`<llvm_source>/include/llvm/Object/ObjectFile.h`) специализируется архитектурно-зависимыми типами объектных файлов, такими как ELF, COFF и MachO. Он способен извлекать символы, смещения и разделы непосредственно из объектов `MemoryBuffer`.

На рис. 7.3 показана диаграмма связей между классами – сплошные стрелки представляют взаимодействия, а пунктирные – наследование.

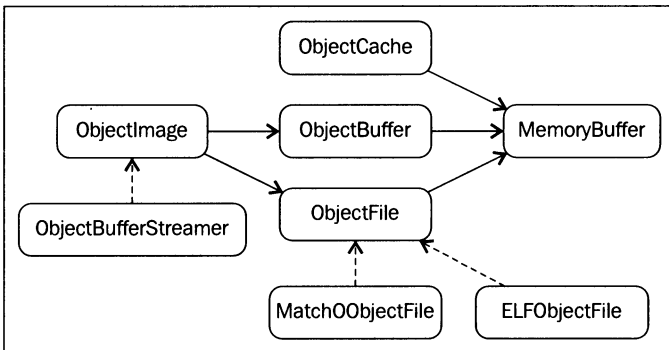


Рис. 7.3. Связи между классами буферов, кэшей и образов

Динамическая компоновка

Объекты загруженных модулей в инфраструктуре MSJIT представлены экземплярами `ObjectImage`. Как упоминалось выше, он имеет прозрачный доступ к буферам в памяти через архитектурно-независимый интерфейс `ObjectFile`. Соответственно, он может обрабатывать символы, разделы и смещения.

Чтобы сгенерировать объекты `ObjectImage`, MSJIT использует средства динамической компоновки, реализованные в классе `RuntimeDyld`. Этот класс предоставляет общедоступный интерфейс к этим средствам, тогда как объекты `RuntimeDyldImpl`, которые специализируются типами объектных файлов, предоставляют фактическую реализацию.

Поэтому метод `RuntimeDyld::loadObject()`, который генерирует объекты `ObjectImage` из `ObjectBuffer`, сначала создает архитектурно-зависимый объект `RuntimeDyldImpl` и затем вызывает `RuntimeDyldImpl::loadObject()`. В ходе этой процедуры также создается объект `ObjectFile`, который затем может быть получен через объект `ObjectImage`. На рис. 7.4 показана диаграмма данного процесса.

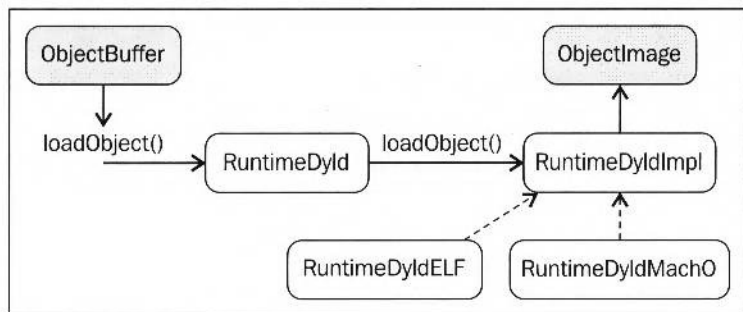


Рис. 7.4. Процесс динамической компоновки

Динамический компоновщик времени выполнения `RuntimeDyld` используется на завершающей стадии создания `Module` для применения смещений и регистрации кадров обработки исключений. Не забывайте, что методы `getFunctionAddress()` и `getPointerToFunction()` механизма выполнения требуют от механизма знания адресов символов (функций). Для решения этой задачи MSJIT также использует компоновщика `RuntimeDyld` и с его помощью определяет адреса символов (вызывая метод `RuntimeDyld::getSymbolLoadAddress()`).

Диспетчер памяти

Класс `LinkingMemoryManager`, еще один наследник класса `RTDyldMemoryManager`, используется механизмом MCJIT как диспетчер памяти. Он сотрудничает с экземпляром `SectionMemoryManager` и перенаправляет запросы ему.

Всякий раз, когда динамический компоновщик `RuntimeDyld` запрашивает адрес символа вызовом метода `LinkingMemoryManager::getSymbolAddress()`, у последнего есть два пути: если символ присутствует в скомпилированном модуле, его адрес извлекается из MCJIT; иначе выполняется попытка получить адрес символа из внешних библиотек, загруженных и представляемых экземпляром `SectionMemoryManager`. Диаграмма на рис. 7.5 иллюстрирует этот механизм. Подробности ищите в реализации `LinkingMemoryManager::getSymbolAddress()` в файле `<llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp`.

Экземпляр `SectionMemoryManager` – это простой диспетчер. Будучи наследником `RTDyldMemoryManager`, класс `SectionMemoryManager` наследует все его методы поиска в библиотеках, но реализует собственные процедуры распределения сегментов кода и данных, опираясь на низкоуровневые единицы `MemoryBlock` (`<llvm_source>/include/llvm/Support/Memory.h`).

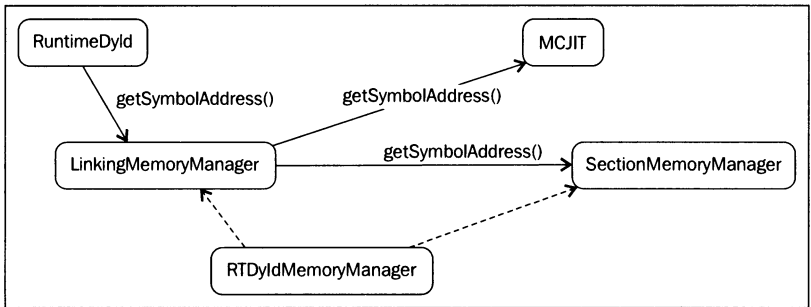


Рис. 7.5. Порядок поиска символов

Эмиссия кода MC

Эмиссия кода MC в инфраструктуре MCJIT выполняется вызовом `MCJIT::emitObject()`. Этот метод выполняет следующие операции:

- создает объект `PassManager`;
- добавляет проход определения информации об архитектуре и вызывает `addPassesToEmitMC()`, чтобы добавить все проходы генератора кода MC;

- выполняет все проходы с помощью метода `PassManager::run()`; получившийся код сохраняет в объекте `Object-BufferStream`;
- добавляет скомпилированный объект в экземпляр `Object-Cache` и возвращает его.

Процесс эмиссии кода в MCJIT является более универсальным, чем в старой инфраструктуре JIT. В отличие от JIT, требующей передачи механизма эмиссии кода и информации о целевой архитектуре, MCJIT прозрачно использует всю информацию из существующей инфраструктуры MC.

Окончательная подготовка модуля

Окончательная подготовка объектов `Module` выполняется методом `MCJIT::finalizeLoadedModules()` и включает: добавление смещений, загруженные модули перемещаются в группу готовых модулей и затем вызывается метод `LinkingMemoryManager::finalizeMemory()`, изменяющий разрешения для страницы памяти. После окончания подготовки объекта, скомпилированные функции готовы к использованию.

Использование механизма MCJIT

Следующий далее файл `sum-mcjit.cpp` содержит все необходимое для динамической компиляции функции `Sum` с использованием инфраструктуры MCJIT вместо JIT. Чтобы подчеркнуть сходство с предыдущим примером динамической компиляции, мы оставили прежний код на месте, а новый заключили в условные инструкции `if`, проверяющие логическую переменную `UseMCJIT`. Поскольку данный пример имеет много общего с примером `sum-jit.cpp`, мы не будем детально описывать фрагменты, о которых уже рассказывалось в обзоре предыдущего примера.

1. Прежде всего нужно подключить заголовочный файл `MCJIT.h`:

```
#include "llvm/ExecutionEngine/MCJIT.h"
```

2. Подключить другие заголовочные файлы и импортировать пространство имен `llvm`:

```
#include "llvm/ADT/OwningPtr.h"
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/ExecutionEngine/JIT.h"
```

```
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/system_error.h"
#include "llvm/Support/FileSystem.h"
using namespace llvm;
```

3. Присвоить переменной `UseMCJIT` значение `true`, чтобы показать, что должна использоваться инфраструктура MCJIT. Чтобы выполнить этот пример с использованием старой инфраструктуры JIT, присвойте переменной значение `false`:

```
bool UseMCJIT = true;

int main() {
    InitializeNativeTarget();
```

4. MCJIT требует инициализировать механизмы парсинга и вывода кода на языке ассемблера:

```
if (UseMCJIT) {
    InitializeNativeTargetAsmPrinter();
    InitializeNativeTargetAsmParser();
}

LLVMContext Context;
std::string ErrorMessage;
OwningPtr<MemoryBuffer> Buffer;

if (MemoryBuffer::getFile("./sum.bc", Buffer)) {
    errs() << "sum.bc not found\n";
    return -1;
}

Module *M = ParseBitcodeFile(Buffer.get(), Context, &ErrorMessage);
if (!M) {
    errs() << ErrorMessage << "\n";
    return -1;
}
```

5. Создать механизм выполнения и вызвать `setUseMCJIT(true)`, чтобы сообщить механизму, что тот должен использовать MCJIT:

```
OwningPtr<ExecutionEngine> EE;
if (UseMCJIT)
    EE.reset(EngineBuilder(M).setUseMCJIT(true).create());
```

```
else
    EE.reset (EngineBuilder (M) .create ());
```

6. Старой инфраструктуре JIT требуется ссылка на `Function`, которая позднее может использоваться для получения указателя на функцию и освобождения занятой памяти:

```
Function* SumFn = NULL;
if (!UseMCJIT)
    SumFn = cast<Function>(M->getFunction("sum"));
```

7. Как упоминалось выше, при работе с инфраструктурой MCJIT рекомендуется вместо метода `getPointerToFunction()` использовать метод `getFunctionAddress()`. Поэтому для каждой инфраструктуры используется свой метод:

```
int (*Sum)(int, int) = NULL;
if (UseMCJIT)
    Sum = (int (*)(int, int)) EE->getFunctionAddress(std::string("sum"));
else
    Sum = (int (*)(int, int)) EE->getPointerToFunction(SumFn);
int res = Sum(4,5);
outs() << "Sum result: " << res << "\n";
res = Sum(res, 6);
outs() << "Sum result: " << res << "\n";
```

8. Так как инфраструктура MCJIT компилирует модули целиком, освобождать память, занимаемую функцией `Sum`, имеет смысл только в старой инфраструктуре JIT:

```
if (!UseMCJIT)
    EE->freeMachineCodeForFunction(SumFn);

llvm_shutdown();
return 0;
}
```

Скомпилировать и скомпоновать `sum-mcjit.cpp` можно командой:

```
$ clang++ sum-mcjit.cpp -g -O3 -rdynamic -fno-rtti $(llvm-config
--cppflags --ldflags --libs jit mcjit native irreader) -o sum-mcjit
```

Можно также использовать измененный `Makefile` из главы 3, «Инструменты и организация». Выполните пример, как показано ниже, и проверьте результаты:

```
$ ./sum-mcjit
Sum result: 9
Sum result: 15
```

Инструменты компиляции LLVM JIT

Проект LLVM содержит несколько инструментов для работы с механизмами динамической компиляции, в том числе: `lli` и `llvm-rtdyld`.

Инструмент `lli`

Инструмент `lli` реализует интерпретатор и JIT-компилятор биткода LLVM, а также использует механизмы выполнения LLVM, описанные выше в этой главе. Давайте поэкспериментируем с этим инструментом, взяв за основу следующий файл (`sum-main.c`) с исходным кодом на языке C:

```
#include <stdio.h>
int sum(int a, int b) {
    return a + b;
}

int main() {
    printf("sum: %d\n", sum(2, 3) + sum(3, 4));
    return 0;
}
```

Инструмент `lli` способен выполнять биткод, если в нем определена функция `main`. Сгенерируйте файл с биткодом `sum-main.bc` следующей командой `clang`:

```
$ clang -emit-llvm -c sum-main.c -o sum-main.bc
```

Выполните биткод с помощью инструмента `lli`, задействовав старую инфраструктуру JIT:

```
$ lli sum-main.bc
sum: 12
```

А теперь, задействовав инфраструктуру MCJIT:

```
$ lli -use-mcjit sum-main.bc
sum: 12
```

Ту же программу можно выполнить с помощью интерпретатора, если указать специальный флаг, который действует значительно медленнее:

```
$ lli -force-interpreter sum-main.bc
sum:12
```


Инструмент *llvm-rtldyld*

Инструмент `llvm-rtldyld` (`<llvm_source>/tools/llvm-rtldyld/llvm-rtldyld.cpp`) – это очень простой инструмент тестирования функций загрузки и компоновки в инфраструктуре MSJIT. Этот инструмент способен читать двоичные объектные файлы с диска и выполнять функции, указанные в командной строке. Он не производит JIT-компиляцию, но позволяет тестировать и выполнять объектные файлы.

Создайте следующие три файла с исходным кодом на языке C: `main.c`, `add.c` и `sub.c`:

- `main.c`

```
int add(int a, int b);
int sub(int a, int b);

int main() {
    return sub(add(3,4), 2);
}
```

- `add.c`

```
int add(int a, int b) {
    return a+b;
}
```

- `sub.c`

```
int sub(int a, int b) {
    return a-b;
}
```

Скомпилируйте их в объектные файлы:

```
$ clang -c main.c -o main.o
$ clang -c add.c -o add.o
$ clang -c sub.c -o sub.o
```

Выполните функцию `main` с помощью инструмента `llvm-rtldyld`, передав ему параметры `-entry` и `-execute`:

```
$ llvm-rtldyld -execute -entry=_main main.o add.o sub.o; echo $?
loaded '_main' at: 0x104d98000
5
```

Имеется также возможность с помощью параметра `-printline` вывести информацию о функциях, скомпилированных с отладочной информацией. Например, взгляните на следующий пример:

```
$ clang -g -c add.c -o add.o
$ llvm-rtldyld -printline add.o
Function: _add, Size = 20
  Line info @ 0: add.c, line:2
  Line info @ 10: add.c, line:3
  Line info @ 20: add.c, line:3
```

В исходных текстах инструмента `llvm-rtldyld` можно видеть примеры абстракции объектов из инфраструктуры MCJIT. Инструмент `llvm-rtldyld` читает указанные объектные файлы в объекты `ObjectBuffer` и генерирует экземпляры `ObjectImage` вызовом `RuntimeDyld::loadObject()`. После загрузки объектных файлов он определяет смещения вызовом `RuntimeDyld::resolveRelocations()`. Затем выясняет точку входа вызовом `getSymbolAddress()` и вызывает функцию.

Кроме того `llvm-rtldyld` использует собственного диспетчера памяти, `TrivialMemoryManager`. Это простой подкласс класса `RTDyldMemoryManager`, разобраться в котором вы легко сможете сами.

Этот замечательный испытательный инструмент поможет вам разобраться с основными понятиями инфраструктуры MCJIT.

Дополнительные ресурсы

Существует множество других источников информации о LLVM JIT в форме электронной документации и примеров использования. В дереве с исходными текстами LLVM, в каталогах `<llvm_source>/examples/HowToUseJIT` и `<llvm_source>/examples/ParallelJIT`, имеются простые примеры исходного кода, которые могут пригодиться при изучении основ механизмов JIT.

В электронном руководстве по LLVM (<http://llvm.org/docs/tutorial/>) имеется отдельная глава, посвященная использованию JIT: <http://llvm.org/docs/tutorial/LangImpl4.html>.

Дополнительную информацию об организации инфраструктуры MCJIT и ее реализации можно также найти по адресу: <http://llvm.org/docs/MCJITDesignAndImplementation.html>.²

² К большому сожалению в Сети крайне мало материалов о LLVM на русском языке и большинство из них безнадежно устарело. Тем не менее, я хотел бы порекомендовать статью «Создание действующего компилятора с помощью инфраструктуры LLVM»: <http://www.ibm.com/developerworks/ru/library/os-creatcompilerllvm1/index.html>. Эта статья тоже содержит устаревшую информацию, но прочитать ее полезно для общего развития. – *Прим. перев.*

В заключение

Динамическая компиляция производится прямо во время выполнения и поддерживается многими виртуальными машинами. В этой главе мы исследовали механизм динамического выполнения LLVM и познакомились с разными его реализациями: старой инфраструктурой JIT и новой MSJIT. Кроме того, мы рассмотрели особенности использования обеих реализаций и на практических примерах попробовали воспользоваться инструментами, использующими механизм динамической компиляции и выполнения.

В следующей главе мы перейдем к исследованию проблемы кросс-компиляции, доступных инструментов и приемов создания кросс-компилятора на основе LLVM.



ГЛАВА 8.

Кросс-платформенная компиляция

Традиционные компиляторы преобразуют исходный код в «родной» выполняемый. В данном случае под словом «родной» подразумевается, что выполняемый код будет работать на той же платформе, что и компилятор, где под платформой подразумевается комбинация аппаратной архитектуры, операционной системы, прикладного двоичного интерфейса (Application Binary Interface, ABI) и системного интерфейса. Эта комбинация определяет механизм взаимодействий между пользовательской программой и системой. Соответственно, если использовать компилятор в GNU/Linux, на компьютере с процессором x86, он сгенерирует выполняемый файл, скомпонованный с соответствующими системными библиотеками и рассчитанный для работы на точно такой же платформе.

Кросс-платформенная компиляция – это процесс использования компилятора с целью создания выполняемых файлов для работы на других платформах, отличающихся от платформы компилятора. Если требуется сгенерировать код, скомпонованный с другими библиотеками, отличающимися от библиотек в системе, где выполняется компиляция, эту задачу обычно можно решить с помощью специальных флагов компилятора. Однако, если целевая платформа (где предполагается выполнять скомпилированный код) отличается от платформы, где производится компиляция, например, аппаратной архитектурой, операционной системой, интерфейсом ABI или форматом объектных файлов, тогда необходимо прибегнуть к кросс-платформенной компиляции.

Возможность кросс-платформенной компиляции особенно важна для разработчиков, создающих приложения для систем с ограниченными ресурсами, например, встраиваемых систем, которые обычно комплектуются низкопроизводительными процессорами и неболь-

шими объемами памяти. А так как компиляция обычно весьма требовательна к производительности CPU и объему памяти, производить ее в таких системах, если и возможно, то весьма затруднительно, так как низкая скорость компиляции будет тормозить процесс разработки. Поэтому в таких случаях кросс-компиляторы оказываются весьма ценными инструментами. В этой главе мы рассмотрим следующие темы:

- ◆ сравнение подходов к кросс-компиляции в Clang и GCC;
- ◆ что такое инструменты компилятора (toolchains);
- ◆ порядок выполнения кросс-платформенной компиляции с помощью Clang;
- ◆ порядок выполнения кросс-платформенной компиляции с помощью нестандартной версии Clang;
- ◆ популярные симуляторы и аппаратные платформы для тестирования выполняемых файлов.

Сравнение GCC и LLVM

Для поддержки кросс-компиляции, компиляторы, такие как GCC, должны собираться со специальными настройками и устанавливаться отдельно для каждой целевой платформы. Обычно имена выполняемых файлов таких компиляторов начинаются с префиксов, определяющих имя целевой архитектуры, как, например `arm-gcc` для архитектуры ARM. Однако, Clang/LLVM позволяет генерировать код для других целевых архитектур простой передачей ключей в командной строке, путей к библиотекам, заголовочным файлам, компоновщику и ассемблеру. Таким образом, единственный драйвер Clang может генерировать код для разных архитектур. Однако, некоторые дистрибутивы LLVM включают не все поддерживаемые архитектуры, например, из соображений уменьшения размеров дистрибутива. С другой стороны, собирая проект LLVM из исходных текстов, можно выбирать, какие целевые архитектуры будут поддерживаться см. главу 1, «Сборка и установка LLVM».

GCC – более старый проект и, соответственно, более зрелый, чем LLVM. Он поддерживает более 50 аппаратных архитектур и широко используется для кросс-компиляции под эти платформы. Однако каждая установка GCC может поддерживать только одну целевую архитектуру. Именно поэтому необходимо устанавливать отдельные версии GCC для каждой целевой архитектуры.

Драйвер Clang, напротив, по умолчанию компилируется и компоуется со всеми целевыми библиотеками. Поэтому компоненты Clang/LLVM всегда имеют доступ к любой архитектурно-зависимой информации, если она потребуется при использовании архитектурно-независимых интерфейсов. Такой подход избавляет от необходимости устанавливать разные версии Clang для каждой целевой архитектуры.

На рис. 8.1 показано, как осуществляется компиляция исходного кода для разных архитектур компиляторами LLVM и GCC. Первый динамически генерирует код для разных процессоров, тогда как второй требует наличия отдельного кросс-компилятора для каждой архитектуры.

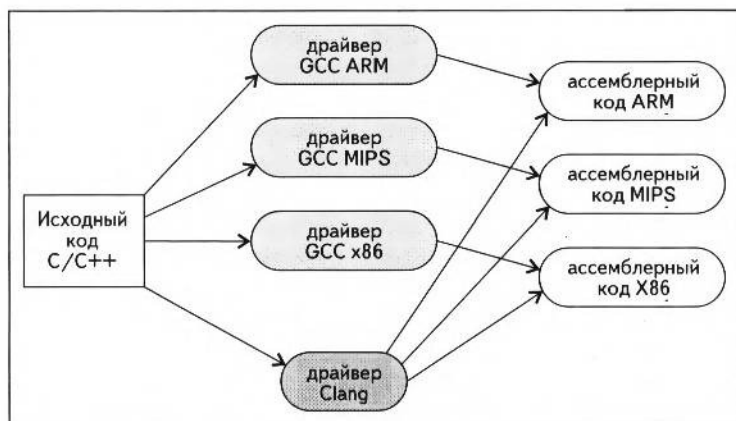


Рис. 8.1. Компиляция для разных архитектур компиляторами LLVM и GCC

При желании можно собрать специализированный драйвер Clang кросс-компилятора, по аналогии с GCC. Хотя в этом случае процедура сборки выглядит сложнее, командная строка такого компилятора будет выглядеть проще. При настройке перед сборкой можно указать фиксированные пути к целевым библиотекам, заголовочным файлам, ассемблеру и компоновщику, и избавиться от необходимости передавать их при каждой кросс-компиляции.

В этой главе мы покажем, как с помощью Clang генерировать код для нескольких платформ, используя параметры командной строки драйвера, и как создать специализированный драйвер кросс-компилятора Clang.

Триады определения целевой архитектуры

Начнем с трех важнейших определений:

- платформа сборки – это платформа, где был собран кросс-компилятор;
- хост-платформа – это платформа, где кросс-компилятор выполняется;
- целевая платформа – это платформа, где должны выполняться программы и библиотеки, сгенерированные кросс-компилятором.

Обычно платформа сборки кросс-компилятора совпадает с хост-платформой. Все три платформы определяются посредством триад. Триады уникально идентифицируют разнообразные целевые архитектуры и включают информацию об аппаратной архитектуре, разновидности и версии операционной системы, типе библиотеки C и формате объектных файлов.

Не существует какого-то строгого формата записи определений триад. Инструменты GNU, к слову, принимают триады, включающие два, три и даже четыре поля в формате: `<arch>-<sys/vendor>-<other>-<other>`, например: `arm-linux-eabi`, `mips-linux-gnu`, `x86_64-linux-gnu`, `x86_64-apple-darwin11` и `sparc-elf`. Clang стремится поддерживать совместимость с GCC и тоже распознает этот формат, но внутренне любые триады преобразуются в единый канонический формат `<arch><sub>-<vendor>-<sys>-<abi>`.

В табл. 8.1 перечислены все возможные значения, поддерживаемые в LLVM, для каждого из полей триад; поле `<sub>` не включено в таблицу, потому что оно представляет варианты архитектур, такие как `v7` в архитектуре `armv7`. Дополнительные подробности о триадах см. в файле `<llvm_source>/include/llvm/ADT/Triple.h`.

Обратите внимание, что не все комбинации `arch`, `vendor`, `sys` и `abi` являются допустимыми. Каждая архитектура поддерживает ограниченное число комбинаций.

Диаграмма на рис. 8.2 иллюстрирует концепцию кросс-компилятора программ для архитектуры ARM, который был собран на платформе `x86`, выполняется на платформе `x86` и генерирует машинные инструкции для процессоров ARM. Многим из вас наверняка будет интересно узнать, что получится, если платформа сборки будет от-

личаться от хост-платформы. Такие комбинации часто называют «канадским крестом» – они чуть сложнее и требуют, чтобы роль компилятора в темном овале не рис. 8.2 играл другой кросс-компилятор. Название «канадский крест» было придумано, когда в Канаде имелись три политические партии, что навело авторов на мысль о сходстве с ситуацией использования трех разных платформ. Без канадского креста не обойтись, например, если вы передаете кросс-компиляторы другим пользователям и обеспечиваете поддержку платформ, отличных от вашей.

Таблица 8.1. Возможные значения полей триад, поддерживаемые в LLVM

Архитектура (<arch>)	Производитель (<vendor>)	Операционная система (<sys>)	Окружение (<abi>)
arm, aarch64, hexagon, mips, mipsel, mips64, mips64el, msp430, ppc, ppc64, ppc64le, r600, sparc, sparcv9, systemz, tce, thumb, x86, x86_64, xc96, nvptx, nvptx64, le32, amdil, spir и spir64	unknown, apple, pc, scei, bgp, bgg, fsl, ibm и nvidia	unknown, auroraux, cygwin, darwin, dragonfly, freebsd, ios, kfreebsd, linux, lv2, macosx, mingw32, netbsd, openbsd, solaris, win32, haiku, minix, rtems, nacl, cnk, bitrig, aix, cuda и nvcl	unknown, gnu, gnuabihi, gnuabi, gnuabi32, eabi, macho, android и elf

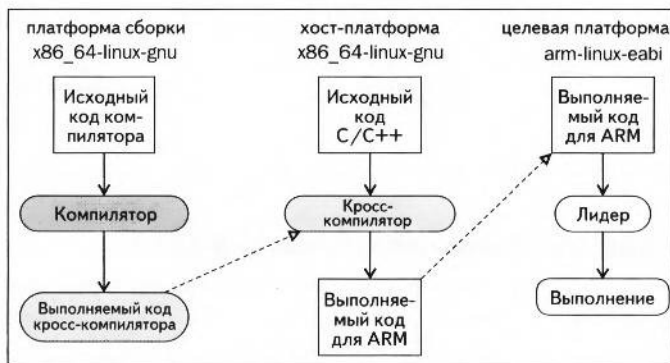


Рис. 8.2. Концепция кросс-компилятора для архитектуры ARM

Подготовка инструментария

Под термином «компилятор» обычно понимается коллекция инструментов для решения задач, так или иначе связанных с компиляцией, таких как анализатор исходного кода (frontend), генератор выполняемого кода (backend), ассемблер и компоновщик. Некоторые из них реализованы в виде отдельных инструментов, другие интегрированы в компилятор. Однако, при разработке приложений или другого ПО, пользователю требуется еще масса других ресурсов, таких как платформо-зависимые библиотеки, отладчик и инструменты для решения вспомогательных задач, таких как чтение объектных файлов. Поэтому производители платформ часто распространяют пакеты инструментов для разработки ПО на своих платформах, которые и называют инструментарием разработчика.

Чтобы сгенерировать или использовать свой кросс-компилятор, важно знать компоненты, составляющие инструментарий, и как они взаимодействуют друг с другом. На рис. 8.3 показаны основные компоненты, необходимые для кросс-компиляции, которые описываются в следующих разделах.

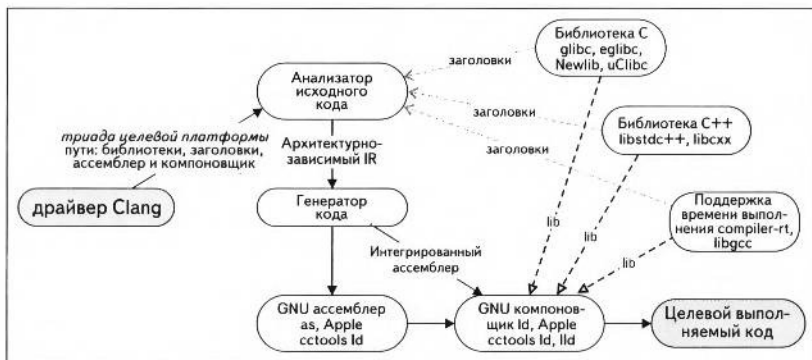


Рис. 8.3. Основные компоненты, необходимые для кросс-компиляции

Стандартные библиотеки C и C++

Библиотека C необходима для поддержки стандартных функциональных возможностей языка C, таких как распределение памяти (`malloc()`/`free()`), обработка строк (`strcmp()`) и ввод/вывод (`printf()`/`scanf()`). В числе наиболее используемых заголовочных файлов библиотеки C можно назвать `stdio.h`,

`stdlib.h` и `string.h`. Существует несколько реализаций библиотеки C. Самыми известными примерами являются библиотека GNU C (`glibc`), `newlib` и `uClibc`. Эти библиотеки доступны для разных платформ и могут переноситься с одной платформы на другую.

Аналогично стандартная библиотека C++ реализует функциональные возможности языка C++, такие как потоковый ввод/вывод, контейнеры, обработка строк и поддержка многопоточного выполнения. В числе примеров реализаций можно назвать GNU `libstdc++` и LLVM `libc++` (см. <http://libcxx.llvm.org>). В действительности полная библиотека GNU C++ включает две библиотеки – `libstdc++` и `libsupc++`. Последняя реализует архитектурно-зависимый уровень, упрощающий перенос между платформами, который связан исключительно с обработкой прерываний и извлечением информации о типах объектов во время выполнения (Run Time Type Information, RTTI). Реализация LLVM `libc++` все еще зависит от сторонних реализаций `libsupc++` для систем, отличных от Mac OS X (дополнительные подробности см. в разделе «Введение в стандартную библиотеку `libc++`» главы 2, «Внешние проекты»).

Кросс-компилятору необходимо знать путь к целевым библиотекам C/C++ и заголовочным файлам, чтобы находить прототипы функций и позднее – правильно скомпоновать библиотеки. Очень важно, чтобы заголовочные файлы соответствовали версии и реализации скомпилированных библиотек. Например, неправильно настроенный кросс-компилятор может по ошибке использовать системные заголовочные файлы и из-за этого найти массу ошибок во время компиляции.

Библиотеки времени выполнения

Для каждой целевой архитектуры должны быть определены специальные функции, имитирующие низкоуровневые операции, которые не поддерживаются самой архитектурой. Например, в 32-разрядных архитектурах обычно отсутствуют 64-разрядные регистры и потому они не имеют возможности обрабатывать 64-разрядные числа непосредственно. Для этой цели могут использоваться 32-разрядные регистры и специальные функции, выполняющие простые арифметические операции (сложение, вычитание, умножение и деление).

Генератор кода порождает вызовы этих функций, предполагая, что ссылки на них будут подставлены во время компоновки. Все необходимые библиотеки должны предоставляться драйвером, а не пользователем. В GCC эта функциональность реализуется библиотекой

времени выполнения `libgcc`. В LLVM имеется аналогичная библиотека `compiler-rt` (см. главу 2, «Внешние проекты»). То есть, драйвер Clang вызывает компоновщика и передает ему ключ `-lgcc` или `-lclang_rt` (чтобы скомпоновать скомпилированный код с библиотекой `compiler-rt`). И снова, очень важно правильно настроить путь к библиотеке времени выполнения, чтобы обеспечить правильную и безошибочную компоновку.

Ассемблер и компоновщик

Ассемблер и компоновщик обычно имеют вид отдельных инструментов и вызываются драйвером компилятора. Например, ассемблер и компоновщик из пакета GNU Binutils поддерживают несколько целевых архитектур, при этом ассемблер и компоновщик для «родной» архитектуры обычно находятся в пути поиска системных библиотек и доступны в виде утилит `as` и `ld`, соответственно. В проекте LLVM имеется свой компоновщик `lld` (<http://lld.llvm.org>), но он все еще находится на экспериментальной стадии.

Для вызова этих двух инструментов к их именам добавляется префикс из триады определения архитектуры и выполняется поиск в каталогах, перечисленных в системной переменной `PATH`. Например, когда генерируется код для архитектуры `mips-linux-gnu`, драйвер может попытаться найти `mips-linux-gnu-as` и `mips-linux-gnu-ld`. Clang способен выполнять поиск по-разному, в зависимости от информации в триаде описания архитектуры.

Для некоторых целевых архитектур не требуется искать внешний ассемблер. Так как проект LLVM включает реализацию прямой эмиссии кода посредством уровня MC, драйвер может использовать ассемблер, интегрированный в MC, если ему передать флаг `-integrated-as`, который по умолчанию считается включенным для некоторых архитектур.

Анализатор исходного кода Clang

В главе 5, «Промежуточное представление LLVM» говорилось, что промежуточное представление LLVM IR, которое генерирует Clang, не является полностью архитектурно-независимым, так же как не является полностью независимым язык C/C++. Кроме того, анализатор исходного кода должен обеспечивать некоторые архитектурно-зависимые ограничения. Следовательно, вы уже должны знать, что несмотря на поддержку той или иной аппаратной архитектуры в Clang,

если триада описания не соответствует в точности этой аппаратной архитектуре, анализатор исходного текста может создать дефектный код LLVM IR, что в свою очередь приведет к несовпадению ABI и ошибкам во время выполнения.

Multilib

Multilib – это решение, позволяющее пользователям выполнять приложения, скомпилированные для разных ABI на одной и той же платформе. Этот механизм позволяет избежать необходимости иметь несколько кросс-компиляторов, при условии, что используемый кросс-компилятор имеет доступ к скомпилированным версиям библиотек и заголовочным файлам для каждого варианта ABI. Например, решение multilib обеспечивает сосуществование библиотек с программной и аппаратной поддержкой вещественной арифметики, то есть библиотек, опирающихся на программную реализацию вещественной арифметики, и библиотек, опирающихся на использование арифметического сопроцессора FPU. Например, GCC имеет несколько версий `libc` и `libgcc` для каждой версии multilib.

В MIPS GCC, например, структура каталогов библиотеки multilib имеет следующий вид:

- `lib/n32`: хранит библиотеки n32, поддерживающие n32 MIPS ABI;
- `lib/n32/EL`: хранит версии `libgcc`, `libc` и `libstdc++` с поддержкой обратного порядка следования байтов;
- `lib/n32/msoft-float`: хранит библиотеки n32 с программной поддержкой вещественной арифметики;
- `lib/n64`: хранит библиотеки n64, поддерживающие n64 MIPS ABI;
- `lib/n64/EL`: хранит версии `libgcc`, `libc` и `libstdc++` с поддержкой обратного порядка следования байтов;
- `lib/n64/msoft-float`: хранит библиотеки n64 с программной поддержкой вещественной арифметики.

Clang поддерживает окружения multilib при условии, что указаны правильные пути к библиотекам и заголовочным файлам. Однако, поскольку на некоторых целевых архитектурах анализатор исходного кода генерирует разные промежуточные представления LLVM IR для разных ABI, всегда тщательно проверяйте пути и триады определения архитектур, чтобы убедиться в их соответствии и избежать появления ошибок во время выполнения.

Кросс-компиляция с аргументами командной строки Clang

Теперь, после знакомства со всеми компонентами инструментария, мы покажем вам, как использовать Clang в роли кросс-компилятора путем передачи соответствующих параметров командной строки.

Примечание. Все примеры в этом разделе проверены на компьютере x86_64, действующем под управлением ОС Ubuntu 12.04. Для загрузки зависимостей мы использовали инструменты, входящие в состав Ubuntu, но команды, имеющие отношение к Clang, должны работать в любых других операционных системах без модификаций (или с незначительными изменениями).

Параметры драйвера, определяющие архитектуру

Динамический выбор триады архитектуры, для которой должен быть сгенерирован код, в Clang используется параметр драйвера `-target=<triple>`. Помимо определения триады можно также использовать другие параметры, позволяющие более точно охарактеризовать целевую архитектуру.

- Параметр `-march=<arch>` определяет базовую архитектуру. Например, `<arch>` может принимать значения: `armv4t`, `armv6`, `armv7` и `armv7f` для ARM, и `mips32`, `mips32r2`, `mips64` и `mips64r2` для MIPS. Этот параметр также определяет базовую архитектуру для генератора объектного кода.
- Конкретный процессор выбирается с помощью параметра `-mcpu=<cpu>`. Например, `cortex-m3` и `cortex-a8` – это конкретные модели процессоров ARM, а `pentium4`, `athlon64` и `corei7-avx2` – модели процессоров семейства x86. Для каждой модели процессора определена базовая архитектура `<arch>`, которая будет использоваться драйвером.
- Параметр `-mfloat-abi=<abi>` определяет тип регистров для хранения вещественных значений: `soft` (программный) или `hard` (аппаратный). Как упоминалось выше, этот параметр отвечает за выбор программной или аппаратной поддерж-

ки вещественной арифметики. Кроме того, он предполагает применение определенных соглашений по вызову и соблюдение других особенностей ABI. Доступны также псевдонимы `-msoftfloat` и `-mhard-float`. Имейте в виду, что если этот параметр не указан, тип ABI выбирается в соответствии с выбранной моделью процессора.

Чтобы увидеть другие возможные параметры, выполните команду `clang --help-hidden`, которая выведет на экран описание даже скрытых параметров.

ЗАВИСИМОСТИ

Для демонстрации возможностей кросс-компиляции в Clang мы скомпилируем программу для архитектуры ARM. Первым шагом установим полный комплект инструментов поддержки ARM и определим имеющиеся компоненты.

Чтобы установить кросс-компилятор GCC для архитектуры ARM с аппаратной поддержкой вещественной арифметики, выполните следующую команду:

```
$ apt-get install g++-4.6-arm-linux-gnueabihf gcc-4.6-arm-linux-gnueabihf
```

Чтобы установить кросс-компилятор GCC для архитектуры ARM с программной поддержкой вещественной арифметики, выполните следующую команду:

```
$ apt-get install g++-4.6-arm-linux-gnueabi gcc-4.6-arm-linux-gnueabi
```

Примечание. Мы только что предложили вам установить полный комплект инструментов GCC, включая кросс-компилятор! Но нужен ли он для использования Clang/LLVM? Как рассказывалось в разделе, посвященном инструментарию, в ходе кросс-компиляции сам компилятор является лишь одним из множества компонентов, включая ассемблер, компоновщик и целевые библиотеки. Для кросс-компиляции необходим инструментарий, подготовленный производителем целевой платформы, потому что только этот инструментарий содержит необходимые заголовочные файлы и библиотеки. Обычно такие комплекты инструментов распространяются вместе с компилятором GCC. Да, мы будем использовать Clang/LLVM для кросс-компиляции, но мы все еще зависим от остальных компонентов инструментария.

При желании можно самостоятельно собрать целевые библиотеки и подготовить свой инструментарий, однако для этого потребуются подготовка образ операционной системы для целевой платформы. Только так вы сможете гарантировать соответствие версий би-

библиотек целевой платформе. Если вы любитель все делать своими руками, мы можем порекомендовать вам замечательное руководство *Cross Linux from Scratch*: <http://trac.clfs.org.1>

Несмотря на то, что утилита `apt-get` автоматически устанавливает все необходимые зависимости, мы все же перечислим основные пакеты, требуемые и рекомендуемые для поддержки кросс-компилятора C++ на основе Clang для архитектуры ARM:

- `libc6-dev-armhf-cross` и `libc6-dev-armel-cross`;
- `gcc-4.6-arm-linux-gnueabi-base` и `gcc-4.6-arm-linux-gnueabi-hf-base`;
- `binutils-arm-linux-gnueabi` и `binutils-arm-linux-gnueabi-hf`;
- `libgcc1-armel-cross` и `libgcc1-armhf-cross`;
- `libstdc++6-4.6-dev-armel-cross` и `libstdc++6-4.6-dev-armhf-cross`.

Кросс-компиляция

Сам кросс-компилятор GCC нас не интересует, но команда, что приводилась в предыдущем разделе, установит все необходимые зависимости, требуемые нашему кросс-компилятору: компоновщик, ассемблер, библиотеки и заголовочные файлы. Теперь мы можем скомпилировать программу `sum.c` (из главы 7, «Динамический компилятор») для платформы `arm-linux-gnueabi-hf` следующей командой:

```
$ clang --target=arm-linux-gnueabi-hf sum.c -o sum
$ file sum
```

```
sum: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs)...
```

Clang найдет все нужные ему компоненты в инструментарии GNU `arm-linux-gnueabi-hf` и сгенерирует выполняемый файл. В этом примере по умолчанию используется архитектура `armv6`, но мы можем более точно определить архитектуру с помощью ключа `--target` и использовать ключ `-mcpu`, чтобы обеспечить еще большую избирательность:

```
$ clang --target=armv7a-linux-gnueabi-hf -mcpu=cortex-a15 sum.c -o sum
```

¹ Похожее руководство на русском языке можно найти по адресу: http://www.opennet.ru/docs/RUS/clfs/CLFS-BOOK-x86_64-64.html. – Прим. перев.

Установка GCC

Триада с определением целевой архитектуры в ключе `--target` используется драйвером Clang для поиска установленной версии GCC с тем же или похожим префиксом. Если будет найдено несколько кандидатов, Clang выберет тот, что точнее соответствует триаде:

```
$ clang --target=arm-linux-gnueabihf sum.c -o sum -v

clang version 3.4 (tags/RELEASE_34/final)
Target: arm--linux-gnueabihf
Thread model: posix
Found candidate GCC installation: /usr/lib/gcc/arm-linux-gnueabihf/4.6
Found candidate GCC installation: /usr/lib/gcc/arm-linux-gnueabihf/4.6.3
Selected GCC installation: /usr/lib/gcc/arm-linux-gnueabihf/4.6
(...)
```

Так как обычно в состав GCC входит ассемблер, компоновщик, библиотеки и заголовочные файлы, Clang будет использовать установленную версию GCC для доступа к требуемым компонентам. Если указать триаду, которая в точности соответствует имени имеющегося в системе инструментария, драйвер Clang легко сможет определить пути к необходимым компонентам. Однако, если указать другую или неполную триаду, драйвер выполнит поиск и выберет инструментарий, наиболее близко соответствующий триаде:

```
$ clang --target=arm-linux sum.c -o sum -v

...
Selected GCC installation: /usr/lib/gcc/arm-linux-gnueabi/4.7
clang: warning: unknown platform, assuming -mfloat-abi=soft
```

Обратите внимание: даже при том, что мы установили две версии инструментария GCC – для `arm-linux-gnueabi` и для `arm-linux-gnueabihf`, драйвер выберет первый. Поскольку в данном примере выбранная платформа неизвестна, предполагается ABI с программной поддержкой вещественной арифметики.

Потенциальные проблемы

Если добавить ключ `-mfloat-abi=hard`, драйвер не будет выводить предупреждение, но по-прежнему будет выбирать архитектуру `arm-linux-gnueabi` вместо `arm-linux-gnueabihf`. Это приведет к появлению ошибок во время выполнения скомпилированной программы, потому что объектный код, скомпилированный с поддержкой аппаратной вещественной арифметики будет скомпонован с библиотекой, включающей программную поддержку:


```
$ clang --target=arm-linux -mfloat-abi=hard sum.c -o sum
```

Причина, почему не выбирается архитектура `arm-linux-gnueabi`, даже если указан ключ `-floatabi=hard`, состоит в том, что мы явно не указали драйверу использовать инструментарий для архитектуры `arm-linux-gnueabi`. Если оставить право решения за драйвером, он выберет первый найденный инструментарий, который может отличаться от желаемого. Этот пример наглядно показывает, что драйвер не всегда выбирает лучший вариант, если не указывать или указывать неполное определение архитектуры, такое как `arm-linux`.

Очень важно знать, какие компоненты входят в инструментарий, чтобы убедиться в правильности выбора. Сделать это можно, например, передав флаг `-###`, который обеспечивает вывод информации об инструментах, используемых драйвером Clang для компиляции, ассемблирования и компоновки программы.

Давайте попробуем передать еще более неопределенное описание целевой архитектуры и посмотрим, что из этого получится. Для этого передадим флаг `--target=arm`:

```
$ clang --target=arm sum.c -o sum
```

```
/tmp/sum-3bbfbc.s: Assembler messages:
/tmp/sum-3bbfbc.s:1: Error: unknown pseudo-op: '.syntax'
/tmp/sum-3bbfbc.s:2: Error: unknown pseudo-op: '.cpu'
/tmp/sum-3bbfbc.s:3: Error: unknown pseudo-op: '.eabi_attribute'
(...)
```

Убрав определение ОС из триады, мы ввели драйвер в заблуждение, что привело к ошибке компиляции. А произошло вот что: драйвер попытался ассемблировать исходный код на языке ассемблера для ARM с использованием ассемблера для «родной» (x86_64) архитектуры. Так как триада с определением целевой архитектуры была весьма не полной и в ней отсутствовало название ОС, инструментарий `arm-linux` не удовлетворил критериям поиска и драйвер выбрал системный ассемблер.

Изменение корневого каталога

Поиск необходимого целевого инструментария выполняется драйвером путем проверки наличия кросс-компиляторов GCC с заданной триадой определения целевой платформы в системных каталогах (см. `<llvm_source>/tools/clang/lib/Driver/ToolChains.cpp`).

В случаях, когда в системе имеется кросс-компилятор не совсем с той триадой или отсутствует вообще, чтобы задействовать имею-

щиеся компоненты, необходимо передать драйверу специальные параметры. Например, параметр `--sysroot` изменяет базовый каталог, где Clang выполняет поиск компонентов инструментария, и может использоваться в ситуациях, когда целевая триада не содержит достаточно информации. Аналогично можно использовать ключ `--gcc-toolchain=<value>`, чтобы указать каталог с требуемым инструментарием.

В инструментарии для архитектуры ARM, установленном в нашей системе, GCC для триады `arm-linux-gnueabi` обнаруживается в каталоге `/usr/lib/gcc/arm-linux-gnueabi/4.6.3`. Начиная с этого каталога, Clang отыскивает все остальные компоненты: библиотеки, заголовочные файлы, ассемблер и компоновщик. Одним из достижимых драйвером каталогов является `/usr/arm-linux-gnueabi`, который содержит следующие подкаталоги:

```
$ ls /usr/arm-linux-gnueabi
```

```
bin include lib usr
```

Компоненты инструментария организованы в этих каталогах точно так же, как «родные» компоненты в каталогах `/bin`, `/include`, `/lib` и `/usr`. Представим, что нам нужно сгенерировать код для `armv7-linux` с процессором `cortex A9`, не уповая, что драйвер найдет нужные компоненты автоматически. Зная, где находятся компоненты `arm-linux-gnueabi`, мы можем передать драйверу флаг `--sysroot`:

```
$ PATH=/usr/arm-linux-gnueabi/bin:$PATH /p/cross/bin/clang  
--target=armv7a-linux --sysroot=/usr/arm-linux-gnueabi -mcpu=cortex-a9  
-mfloat-abi=soft sum.c -o sum
```

Напомним еще раз, что такой подход может пригодиться при наличии необходимых компонентов, но в отсутствие соответствующей версии GCC. Есть три основные причины, объясняющие, почему этот подход дает положительные результаты.

- Триада `armv7a-linux` активирует компиляцию для архитектуры ARM и операционной системы `linux`. Кроме всего прочего она сообщает драйверу, что тот должен использовать синтаксис вызова ассемблера и компоновщика GNU. Если название ОС опустить, Clang по умолчанию использует синтаксис ассемблера `Darwin`, что приведет к появлению ошибок компиляции.
- По умолчанию компилятор выполняет поиск библиотек и заголовочных файлов в каталогах `/usr`, `/lib` и `/usr/include`.

Параметр `--sysroot` переопределяет поведение драйвера по умолчанию и вынуждает его выполнять поиск этих каталогов не в корневом каталоге системы, а в `/usr/arm-linux-gnueabi`.

- Переменная окружения `RPATH` изменяется, чтобы исключить возможность использования версий `as` и `ld` по умолчанию. Этим мы вынуждаем драйвер сначала выполнить поиск в `/usr/arm-linux-gnueabi/bin`, где находятся версии `as` и `ld` для архитектуры ARM.

Создание кросс-компилятора Clang

Clang поддерживает компиляцию кода для любой целевой архитектуры, как было показано в предыдущем разделе. Однако, иногда бывает желательно собрать специализированную версию кросс-компилятора Clang, и вот почему:

- когда пользователь не желает писать длинные командные строки для вызова драйвера;
- когда производитель хочет распространять инструментарий Clang для определенной архитектуры;

Параметры настройки

Система конфигурирования LLVM предлагает следующие параметры для настройки сборки кросс-компиляторов.

- `--target`: определяет триаду с описанием архитектуры по умолчанию, для которой кросс-компилятор Clang будет генерировать код. Этот параметр тесно связан с понятиями целевой платформы, хост-платформы и платформы сборки, представленными выше. Имеются также параметры `--host` и `--build`, но значения для них обычно определяются сценарием `configure` – оба устанавливаются в соответствии с «родной» платформой.
- `--enable-targets`: определяет, какие целевые архитектуры будут поддерживаться данным кросс-компилятором. Если не указан, включается поддержка всех архитектур. Помните, что для выбора других архитектур, отличных от архитектуры по умолчанию, вам потребуется использовать параметр команд-

ной строки `--target` с целевыми архитектурами, как описывалось выше.

- `--with-c-include-dirs`: определяет список каталогов, где кросс-компилятор должен искать заголовочные файлы. Благодаря этому параметру можно избежать необходимости указывать многочисленные ключи `-I` для организации поиска заголовочных файлов, которые могут находиться в нестандартных каталогах. Указанные здесь каталоги просматриваются первыми, и только в случае неудачи поиск будет выполняться в системных каталогах.
- `--with-gcc-toolchain`: определяет целевой инструментарий GCC, уже имеющийся в системе. Местоположение компонентов, определяемое этим параметром, будет «жестко зашито» в кросс-компилятор в виде параметра `--gcc-toolchain`.
- `--with-default-sysroot`: добавляет параметр `--sysroot` во все вызовы компилятора, вызываемого кросс-компилятором.

Перечень всех параметров настройки LLVM/Clang можно получить командой `<llvm_source>/configure --help`. Для включения поддержки архитектурно-зависимых особенностей можно использовать дополнительные параметры настройки (скрытые), такие как `--with-cpu`, `--with-float`, `--with-abi` и `--with-fpu`.

Сборка и установка кросс-компилятора на основе Clang

Настройка, сборка и установка кросс-компилятора мало чем отличаются от традиционного способа сборки LLVM и Clang, описанного в главе 1, «Сборка и установка LLVM».

Поэтому, если предположить, что все исходные тексты на месте, собрать кросс-компилятор LLVM для архитектуры ARM Cortex-A9 можно следующей командой:

```
$ cd <llvm_build_dir>
$ <PATH_TO_SOURCE>/configure --enable-targets=arm --disable-optimized
--prefix=/usr/local/llvm-arm --target=armv7a-unknown-linux-gnueabi

$ make && sudo make install
$ export PATH=$PATH:/usr/local/llvm-arm
$ armv7a-unknown-linux-gnueabi-clang sum.c -o sum
$ file sum
```

```
sum: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs)...
```

Как уже говорилось в разделе «Триады определения целевой архитектуры», триада определения целевой архитектуры может содержать до четырех элементов, но некоторые инструменты принимают более короткие триады. Сценарий `configure` для проекта LLVM, который генерируется с помощью GNU autotools, ожидает получить полную триаду, со всеми четырьмя элементами, – с информацией о производителе во втором элементе. Так как для нашей платформы не определен какой-то конкретный производитель, мы расширим нашу триаду до `armv7a-unknown-linux-gnueabi`. Если в данном случае указать триаду с тремя элементами, сценарий `configure` потерпит неудачу.

Никаких других дополнительных параметров для определения инструментария не требуется, потому что Clang выполняет поиск требуемой версии GCC, как обычно.

Допустим, что вы скомпилировали и установили дополнительные заголовочные файлы и библиотеки для архитектуры ARM в каталоги `/opt/arm-extra-libs/include` и `/opt/arm-extra-libs/lib`, соответственно. Указав флаг `--with-c-include-dirs=/opt/arm-extra-libs/include`, вы сможете добавить этот каталог в путь поиска заголовочных файлов на постоянной основе; однако для корректной компоновки вам все еще будет необходимо указывать флаг `-L/opt/arm-extra-libs/lib`.

```
$ <PATH_TO_SOURCE>/configure --enable-targets=arm --disable-optimized
--prefix=/usr/local/llvm-arm --target=armv7a-unknown-linux-gnueabi
--with-c-include-dirs=/opt/arm-extra-libs/include
```

Аналогично можно определить корневой каталог (флаг `--sysroot`) и путь поиска инструментария GCC (флаг `--with-gcc-toolchain`) для постоянного использования драйвером. Это избыточно для выбранной триады ARM, но может пригодиться для других целевых архитектур:

```
$ <PATH_TO_SOURCE>/configure --enable-targets=arm --disable-optimized
--prefix=/usr/local/llvm-arm --target=armv7a-unknown-linux-gnueabi
--with-gcc-toolchain=arm-linux-gnueabi
--with-default-sysroot=/usr/arm-linux-gnueabi
```

Альтернативные методы сборки

Существуют также другие инструменты, с помощью которых можно собрать инструментарий на основе LLVM/Clang, и другие системы сборки, поддерживаемые в LLVM. Одним из альтернативных решений является создание обертки для упрощения процесса.

Ninja

Одним из альтернативных способов сборки кросс-компилятора является использование CMake и Ninja. Ninja – это проект маленькой и быстрой системы сборки.

Вместо выполнения традиционных этапов настройки и сборки, можно воспользоваться специальными возможностями CMake и создать инструкции сборки для Ninja, которые затем будут выполнять сборку и установку кросс-компилятора для требуемой целевой архитектуры.

Инструкции и описание можно найти по адресу: <http://llvm.org/docs/HowToCrossCompileLLVM.html>.

ELLCC

Инструмент ELLCC – это фреймворк на основе LLVM, который используется при создании инструментариев для встраиваемых архитектур.

Его целью является упрощение процедуры создания и использования кросс-компиляторов. Он легко расширяется, поддерживает новые целевые архитектуры и удобен для разработчиков, создающих программы для разных платформ.

Кроме того, ELLCC компилирует и устанавливает несколько дополнительных компонентов инструментария, включая отладчик и QEMU для тестирования платформ.

Инструмент есс – это кросс-компилятор, готовый к использованию. Он действует, опираясь на кросс-компиляторы Clang, и принимает параметры командной строки, совместимые с GCC и Clang, для компиляции под любые поддерживаемые целевые архитектуры. Более подробную информацию о нем можно получить по адресу: <http://ellcc.org/>.

EmbToolkit

Инструментарий для встраиваемых систем (Embedded system Toolkit) – еще один фреймворк для создания средств компиляции для встраиваемых систем. Поддерживает создание инструментариев на основе Clang или LLVM, компилируя компоненты и предоставляя корневую файловую систему одновременно.

Поддерживает интерфейсы ncurses и GUI для выбора компонентов. Более подробную информацию можно получить по адресу: <https://www.embtoolkit.org/>.

Тестирование

Наиболее разумный способ убедиться в успехе кросс-компиляции – запустить скомпилированную программу на целевой платформе. Однако, когда такая платформа недоступна, можно воспользоваться специализированными симуляторами.

Одноплатные компьютеры

Существует множество разновидностей одноплатных компьютеров (development boards) для разных платформ. В настоящее время многие из них можно купить в Интернете. Например, можно без труда найти одноплатные компьютеры ARM, оснащенные разными процессорами, от простых Cortex-M до многоядерных Cortex-A.

Оснащение периферийными устройствами может отличаться, но обычно одноплатные компьютеры снабжаются устройствами Ethernet, Wi-Fi, USB и картами памяти. Соответственно, скомпилированные приложения можно загружать в такие компьютеры по сети, через USB или записывать на флешку и выполнять на «голом железе» или во встроенной системе Linux/FreeBSD.

В табл. 8.2 перечислены примеры одноплатных компьютеров.

Таблица 8.2. Примеры одноплатных компьютеров

Название	Особенности	Архитектура/ процессор	Ссылка
Panda Board	Linux, Android, Ubuntu	ARM, Dual Core Cortex A9	http://pandaboard.org/
Beagle Board	Linux, Android, Ubuntu	ARM, Cortex A8	http://beagleboard.org/
SEAD-2	Linux	MIPS M14K	http://www.timesys.com/ supported/processors/mips
Carambola-2	Linux	MIPS 24K	http://8devices.com/ carambola-2

Существует также огромное разнообразие мобильных устройств с процессорами ARM и MIPS, действующими под управлением ОС Android. Вы можете попробовать использовать Clang на них.

Симуляторы

Производители часто разрабатывают симуляторы для своих процессоров, потому что цикл разработки программного обеспечения начинается еще до того, как аппаратная платформа будет готова. Инструментальные наборы с симуляторами передаются клиентам или используются внутри компаний-производителей для тестирования продуктов.

Такие симуляторы, созданные производителями, являются одним из способов протестировать программу, скомпилированную кросс-компилятором. Однако, существует также несколько открытых проектов эмуляторов для разных архитектур и процессоров. QEMU – открытый проект эмулятора, поддерживающий два режима эмуляции – пользовательский и системный.

В пользовательском режиме QEMU позволяет выполнять на текущей платформе отдельные процессы, скомпилированные для целевой платформы. Например, программа, скомпилированная и скомпонованная для архитектуры ARM, как описывается в предыдущем разделе, почти наверняка будет работать под управлением ARM-QEMU, действующем в пользовательском режиме.

В системном режиме эмулятор воспроизводит поведение целой системы, включая использование периферии и множества процессоров. Поскольку в этом случае эмулируется полный процесс загрузки, необходима операционная система. Существуют варианты эмуляции одноплатных компьютеров в QEMU. Этот продукт также идеально подходит для тестирования программного обеспечения, выполняющегося без поддержки операционной системы или взаимодействующего с периферией.

QEMU поддерживает такие архитектуры, как ARM, MIPS, OpenRISC, SPARC, Alpha и MicroBlaze с разными семействами процессоров. Более подробную информацию можно получить по адресу: <http://qemu-project.org>.

Дополнительные ресурсы

Официальная документация для Clang содержит массу полезной информации об использовании Clang в роли кросс-компилятора, например: <http://clang.llvm.org/docs/CrossCompilation.html>.

В заключение

Кросс-компиляторы – важный ресурс для разработки программных продуктов под другие платформы. Clang специально проектировался так, чтобы обеспечить свободу кросс-компиляции, которая может выполняться драйвером динамически.

В этой главе мы познакомили вас с элементами, образующими среду кросс-компиляции, и рассказали, как Clang взаимодействует с ними при создании выполняемых файлов. Мы также показали, в каких ситуациях может пригодиться кросс-компилятор Clang и дали несколько рекомендаций по сборке, установке и использованию кросс-компилятора.

В следующей главе мы представим статический анализатор Clang и покажем, как искать типичные ошибки в больших объемах исходного кода.



ГЛАВА 9.

Статический анализатор Clang

Человеку трудно планировать разработку абстрактного аппарата из-за отсутствия простого способа оценить его сложность и трудозатраты на его создание. Неудивительно, что в истории развития программной индустрии так много неудач, обусловленных недооценкой сложности. Если разработка программного продукта требует особого внимания к координации и организации труда, то дальнейшее сопровождение этого продукта оказывается еще более сложной задачей.

Чем старше программное обеспечение, тем сложнее его сопровождать. Обычно это обусловлено разностью взглядов разных поколений программистов. Когда новый программист приступает к сопровождению старого программного продукта, часто возникает стремление отделить малопонятные участки кода, изолировать их и превратить в отдельную, неприкасаемую библиотеку.

Такие сложные проекты требуют применения новой категории инструментов, помогающих программистам находить причины появления непонятных ошибок. Главная задача статического анализатора Clang – помочь автоматизировать анализ больших объемов кода и выявление наиболее распространенных видов ошибок, допускаемых в программах на C, C++ или Objective-C до компиляции. В этой главе мы рассмотрим следующие темы:

- ♦ чем отличаются обычные предупреждения компилятора от предупреждений статического анализатора Clang;
- ♦ как использовать Clang в простых проектах;
- ♦ как с помощью инструмента `scan-build` проверять проекты большого объема;
- ♦ как расширять статический анализатор Clang собственными средствами проверки ошибок.

Роль статического анализатора

В общей структуре LLVM, проект связан с анализатором исходного кода Clang, если действует на уровне исходного кода (C/C++) из-за того, что восстановление исходной информации из LLVM IR является более сложной задачей. Одним из наиболее интересных инструментов на основе Clang является статический анализатор Clang Static Analyzer – проект, использующий множество **средств проверки (checkers)** для вывода сообщений об ошибках, подобных традиционным предупреждениям компилятора. Каждое средство проверки определяет соответствие определенному правилу.

Подобно классическим предупреждениям, сообщения статического анализатора помогают программисту находить ошибки на ранних этапах разработки, не откладывая эту работу в долгий ящик. Анализ выполняется после парсинга, но перед дальнейшей компиляцией. С другой стороны, инструмент может потребовать значительного времени для анализа, и именно поэтому он редко интегрируется в процесс компиляции. Например, статический анализатор может потратить несколько часов на анализ всех исходных текстов LLVM.

У статического анализатора Clang имеется как минимум два соперника, инструменты Fortify и Coverity. Первый разрабатывает Hewlett Packard (HP), а второй – Synopsis. Оба инструмента имеют свои сильные и слабые стороны, но только Clang является открытым проектом, что дает нам возможность заглянуть в его исходные тексты, чтобы понять, как он действует, что мы и сделаем в этой главе.

Сравнение классического механизма предупреждений со статическим анализатором Clang

Алгоритм, используемый в статическом анализаторе Clang, имеет **экспоненциальную временную сложность**, то есть, время, необходимое для анализа единицы программы, возрастает по экспоненциальному закону с ростом объема этой единицы. В данный алгоритм, как и во многие другие подобные алгоритмы, введены *ограничения*, с целью уменьшить время его работы и потребление памяти, однако этого недостаточно, чтобы привести временную сложность хотя бы к полиномиальной.

Экспоненциальная природа инструмента объясняет одно из самых существенных ограничений: он способен анализировать не более одной

единицы компиляции за раз и не выполняет межмодульные виды анализа. Тем не менее, это очень ценный инструмент, потому что опирается на **механизм символического выполнения** (symbolic execution engine).

Прежде чем показать пример, как механизм символического выполнения способен помогать программистам в поисках странных ошибок, мы хотим показать фрагмент кода с очень простой ошибкой, которая легко определяется большинством компиляторов:

```
#include <stdio.h>
void main() {
    int i;
    printf ("%d", i);
}
```

Здесь используется *неинициализированная* переменная, из-за чего программа выводит непредсказуемые значения, так как они зависят от того, что хранилось в памяти на момент запуска программы. Очевидно, что простая автоматизированная проверка может сэкономить немало времени и нервов при отладке.

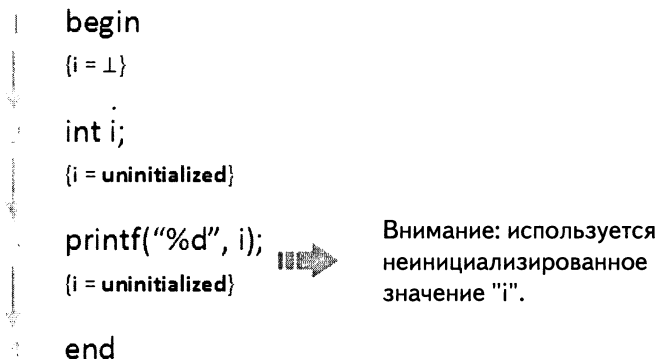
Знакомые с методами анализа, применяемыми в компиляторах, наверняка заметят, что такую проверку можно реализовать с применением *прямого анализа потока данных* (forward dataflow analysis), в котором *оператор слияния* выполняет *объединение*, для распространения состояния каждой переменной и определения – была ли она инициализирована. Прямой анализ потока данных распространяет информацию о состоянии переменных в каждом базовом блоке, начиная с первого базового блока функции и передавая ее во все последующие базовые блоки. Оператор слияния определяет, как объединять информацию, поступающую из нескольких предшествующих базовых блоков. Оператор слияния, действующий как *объединение*, припишет базовому блоку результат объединения множеств состояний переменных всех предшествующих блоков.

Если механизм анализа достигнет строки, где используется неинициализированная переменная, он должен вывести предупреждение. Для этого наша инфраструктура потока данных присваивает каждой переменной в программе следующие состояния:

- символ \perp , когда о состоянии переменной ничего не известно (неизвестное состояние);
- метка **initialized** (инициализирована), когда известно, что переменная была инициализирована;
- метка **uninitialized** (неинициализирована), когда известно, что переменная не была инициализирована;

- символ \perp , когда переменная может быть инициализирована или неинициализирована (то есть, когда нет уверенности).

На рис. 9.1 показана диаграмма анализа потока данных в только что представленной простой программе на С.



\perp = unknown state

Рис. 9.1. Диаграмма анализа потока данных в простой программе на С

Как видите, информация о состоянии переменной легко распространяется по строкам кода. Когда механизм анализа достигает строки с вызовом функции `printf`, он обнаруживает, что в этой строке используется переменная `i`, проверяет – что известно о ее состоянии, выясняет, что она неинициализирована и выводит предупреждение.

Так как этот алгоритм имеет полиномиальную временную сложность, он действует достаточно быстро.

Чтобы увидеть, как простой анализ может терять точность, рассмотрим код, написанный программистом Джо, известным своим искусством вносить странные ошибки. Джо легко может обмануть наш механизм анализа и спрятать истинное состояние переменной за нагромождением разных путей выполнения программы.

```

#include <stdio.h>
void my_function(int unknown_value) {
    int schroedinger_integer;
    if (unknown_value)
        schroedinger_integer = 5;
    printf("hi");
    if (!unknown_value)

```

```
    printf("%d", schroedinger_integer);
}
```

Теперь посмотрим, как инфраструктура анализа потока данных вычислит состояния переменных в этой программе (рис. 9.2).

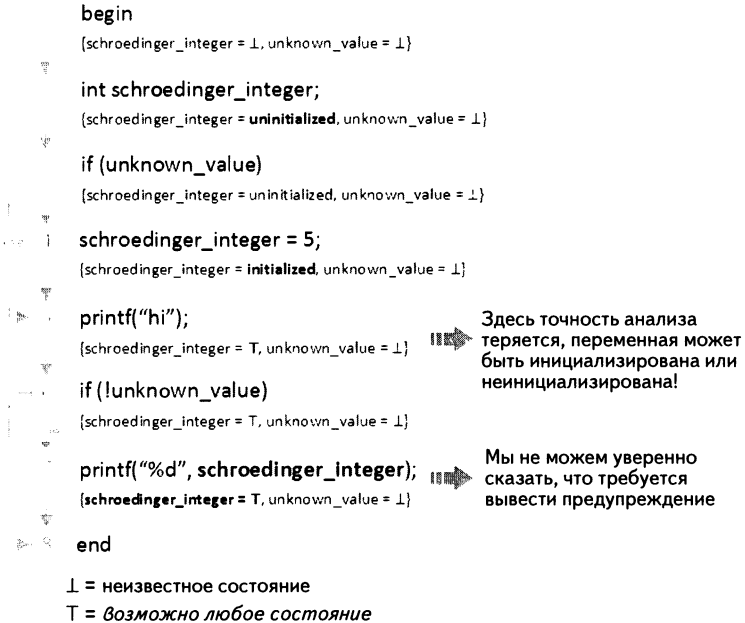


Рис. 9.2. Определение состояния переменных в программе с несколькими путями выполнения

Как показано на диаграмме, в узле 4 переменная инициализируется первый раз (выделена жирным). Однако к узлу 5 ведут два разных пути: ветки true и false инструкции if в узле 3. В одном случае переменная `schroedinger_integer` инициализируется, а в другом нет. Оператор слияния определяет, как должны объединяться предыдущие результаты. В данном случае он действует как оператор объединения (union) и пытается сохранить оба варианта данных, объявив `schroedinger_integer` как T (возможно любое состояние).

Проверяя узел 7, где используется переменная `schroedinger_integer`, механизм анализа не в состоянии определить, является ли такое состояние переменной следствием ошибки в коде. Иными словами, он «запутался» в напластованиях состояний. Наш простой

механизм может попробовать предупредить человека, что значение может быть неинициализировано, и в данном случае это будет правильно. Однако, если последнюю проверку условия в коде заменить на `if (unknown_value)`, предупреждение не будет соответствовать действительности, потому что теперь оно соответствует пути, в котором переменная `schroedinger_integer` действительно инициализируется.

Такая потеря точности в механизме анализа обусловлена тем, что инфраструктура потоков данных не различает разные пути выполнения и не в состоянии точно смоделировать происходящее при всех возможных условиях.

Подобные ложные срабатывания крайне нежелательны, потому что запутывают программиста ложными предупреждениями там, где программный код не имеет ошибок, и «заслоняют» предупреждения о настоящих ошибках. В действительности, если механизм анализа сгенерирует даже малое число ложных предупреждений, программист почти наверняка проигнорирует все предупреждения.

Возможности механизма символического выполнения

Механизм символического выполнения приходит на выручку там, где простой анализ потоков данных не обладает всей полнотой информации о программе. Он строит граф достижимых состояний программы и имеет возможность учитывать все пути выполнения программного кода. Напомним, что в процессе отладки программы вы, фактически, исследуете только один путь выполнения. При отладке с применением мощной виртуальной машины, такой как `valgrind`, с целью найти возможные утечки памяти, она также исследует только один путь выполнения.

Напротив, механизм символического выполнения способен исследовать все возможные пути, фактически не выполняя код. Это очень мощная особенность, но она требует значительного времени для обработки программ.

Как и при анализе потоков данных, механизм символического выполнения присваивает начальные значения всем переменным, какие только обнаружит при обходе программы в порядке выполнения кода. Отличия начинают проявляться по достижении конструкций управления потоком выполнения: механизм символического выполнения разбивает путь выполнения на две ветви и продолжает анализировать

каждый путь в отдельности. Этот граф называется *графом достижимых состояний* программы, а на рис. 9.3 показан простой пример, иллюстрирующий процесс анализа программы Джо.

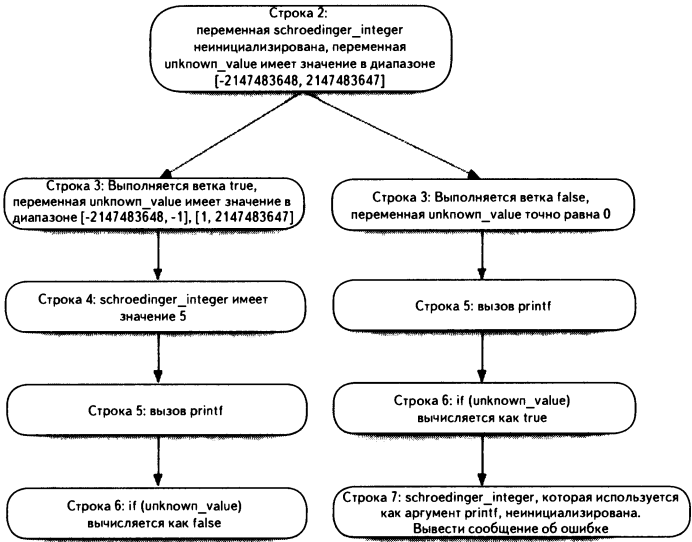


Рис. 9.3. Процесс анализа программы Джо механизмом символического выполнения

В этом примере первая инструкция `if` (в строке 6) делит граф достижимых состояний на два пути: в одном из них `unknown_value` имеет ненулевое значение, а в другом `unknown_value` равна нулю. С этого момента механизм оперирует важным ограничением на основе `unknown_value` и будет использовать его при обработке других веток.

Следуя этой стратегии, механизм символического выполнения приходит к заключению, что левая ветвь (см. рис. 9.3) нигде не использует переменную `schroedinger_integer`, даже при том, что в ней этой переменной присваивается значение 5. При исследовании правой ветви, напротив, обнаруживается, что `schroedinger_integer` передается в вызов функции `printf()` как параметр. Однако в этой ветви переменная оказывается неинициализирована перед использованием. Используя такой граф, можно точно утверждать, что в коде имеется ошибка.

Давайте сравним граф достижимых состояний программы с **графом потока выполнения (Control Flow Graph, CFG)** для того же

самого программного кода, наряду с типичными «рассуждениями» механизма анализа потока данных (см. рис. 9.4).

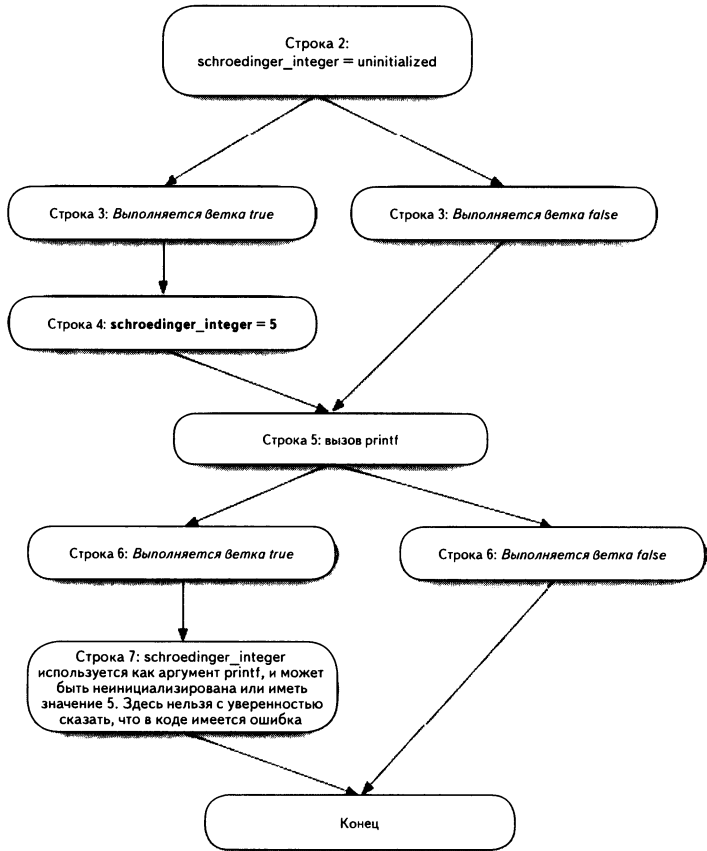


Рис. 9.4. Граф потока выполнения

Первое, что бросается в глаза, граф потока управления (CFG) может ветвиться, отражая изменение направления потока управления, но в нем также имеются точки слияния ветвей, что исключает «комбинаторный взрыв», наблюдаемый в графе достижимых состояний программы. В точках слияния ветвей анализ потока данных может использовать оператор слияния, действующий как объединение или пересечение, чтобы скомбинировать информацию, поступающую из разных ветвей (узел в строке 5). Если слияние выполняется пу-

тем объединения, механизм анализа приходит к заключению, что `schroedinger_integer` может быть инициализирована или иметь значение 5, как в нашем последнем примере. Если слияние выполняется путем пересечения, оказывается, что информация о переменной `schroedinger_integer` отсутствует (неизвестное состояние).

Необходимость объединять данные при типичном анализе потока данных является серьезным ограничением, которое отсутствует в механизме символического выполнения. Благодаря этому механизм символического выполнения способен получать намного более точные результаты, как если бы вы выполнили тестирование программы с разными исходными данными, хотя и ценой больших затрат времени и памяти.

Тестирование статического анализатора

В этом разделе мы покажем, как применять статический анализатор Clang на практике.

Использование драйвера и компилятора

Приступая к использованию статического анализатора, помните, что команда `clang -ccl` вызывает компилятор, а команда `clang` вызывает драйвер компилятора. Драйвер отвечает не только за общую организацию работы всех программ LLVM, вовлеченных в процесс компиляции, но и за передачу адекватных параметров с информацией о вашей системе.

Многие разработчики предпочитают вызывать компилятор непосредственно. Но иногда это может вызывать затруднения из-за незнания, где находятся заголовочные файлы или других конфигурационных параметров, которые известны только драйверу Clang. С другой стороны, компилятор позволяет указывать параметры, необходимые только разработчикам и позволяющие отлаживать скомпилированные программы. Давайте посмотрим, как воспользоваться компилятором и драйвером, чтобы проверить единственный файл с исходным кодом.

Компилятор `clang -ccl -analyze -analyzer-checker=<package> <file>`

Драйвер `clang --analyze -Xanalyzer -analyzer-checker=<package> <file>`

Здесь тег `<file>` обозначает имя файла с исходным кодом, который требуется проанализировать, а тег `<package>` позволяет выбрать конкретную коллекцию заголовочных файлов.

Обратите внимание, что при запуске драйвера статический анализ запускается флагом `--analyze`. Флаг `-xanalyzer` обеспечивает передачу следующего за ним флага непосредственно компилятору, что дает возможность передавать произвольные флаги. Так как драйвер является всего лишь промежуточным звеном, во всех последующих примерах мы будем использовать компилятор непосредственно. Кроме того, для таких простых примеров, как наши, пользоваться компилятором совсем несложно. Однако, если вам потребуется драйвер, чтобы задействовать средства проверки официальным способом, не забывайте добавлять флаг `-xanalyzer` перед каждым флагом, который требуется передать компилятору.

Получение списка доступных средств проверки

Средство проверки (`checker`) – это единица анализа, которую статический анализатор может выполнить в отношении вашего кода. Каждый вид анализа пытается найти ошибки определенного типа. Статический анализатор позволяет выбирать все имеющиеся средства анализа или из подмножество.

Если вы до сих пор не установили Clang, обращайтесь к главе 1, «Сборка и установка LLVM», за инструкциями по установке. Получить список установленных средств проверки можно с помощью следующей команды:

```
$ clang -ccl -analyzer-checker-help
```

Она выведет длинный список доступных средств проверки, которые вы получаете вместе с Clang «из коробки». Теперь рассмотрим вывод команды `-analyzer-checker-help`:

```
OVERVIEW: Clang Static Analyzer Checkers List
```

```
USAGE: -analyzer-checker <CHECKER or PACKAGE,...>
```

```
CHECKERS:
```

```
alpha.core.BoolAssignment Warn about assigning non-{0,1} values  
to Boolean variables
```

Имена средств проверки следуют канонической форме `<package> . <subpackage> . <checker>`, что упрощает использование определенных наборов взаимосвязанных видов анализа.

В табл. 9.1 приводится список наиболее важных пакетов, а также некоторые средства проверки, входящие в эти пакеты.

Таблица 9.1. Наиболее важные пакеты средств проверки

Пакет	Содержит	Примеры
alpha	Средства проверки, в настоящий момент находящиеся в стадии разработки	<code>alpha.core.BoolAssignment</code> , <code>alpha.security.MallocOverflow</code> и <code>alpha.unix.cstring.NotNullTerminated</code>
core	Основные, универсальные средства проверки	<code>core.NullDereference</code> , <code>core.DivideZero</code> , и <code>core.StackAddressEscape</code>
cplusplus	Единственное средство проверки для операций распределения памяти в C++ (остальные пока находятся в пакете alpha)	<code>cplusplus.NewDelete</code>
debug	Средства проверки для вывода отладочной информации статического анализатора	<code>debug.DumpCFG</code> , <code>debug.DumpDominators</code> и <code>debug.ViewExplodedGraph</code>
llvm	Единственное средство проверки, выявляющее – следует ли исходный код стандартам оформления LLVM	<code>llvm.Conventions</code>
osx	Средства проверки для программ, написанных специально для Mac OS X	<code>osx.API</code> , <code>osx.cocoa.ClassRelease</code> , <code>osx.cocoa.NonNilReturnValue</code> и <code>osx.coreFoundation.CFError</code>
security	Средства проверки для кода, который может стать источником уязвимостей	<code>security.FloatLoopCounter</code> , <code>security.insecureAPI.UncheckedReturn</code> , <code>security.insecureAPI.gets</code> и <code>security.insecureAPI.strcpy</code>
unix	Средства проверки для программ, написанных специально для UNIX	<code>unix.API</code> , <code>unix.Malloc</code> , <code>unix.MallocSizeof</code> и <code>unix.MismatchedDeallocator</code>

Давайте проверим код Джо, способный обманывать простой анализ, который выполняется большинством компиляторов. Сначала попробуем использовать классический подход. Для этого просто вызовем драйвер Clang и предложим ему выполнить проверку синтаксиса:

```
$ clang -fsyntax-only joe.c
```

Флаг `syntax-only` обеспечивает вывод предупреждений, при обнаружении синтаксических ошибок. Но эта проверка не обнаруживает в примере программы ничего подозрительного.

Теперь попробуем применить механизм символического выполнения:

```
$ clang -cc1 -analyze -analyzer-checker=core joe.c
```

Если эта команда потребует от вас указать каталог с заголовочными файлами, воспользуйтесь драйвером, как показано ниже:

```
$ clang --analyze -Xanalyzer -analyzer-checker=core joe.c
```

```
./joe.c:10:5: warning: Function call argument is an uninitialized value
printf(«%d», schroedinger_integer);
      ^~~~~~
```

```
1 warning generated.
```

Точно в яблочко! Напомним, что флагу `analyzer-checker` требуется передавать полное имя средства проверки или имя пакета. Мы решили использовать пакет `core` целиком, но точно так же можно было бы использовать только одно средство проверки `core.CallAndMessage`, проверяющее параметры в вызовах функций.

Обратите внимание, что все команды статического анализатора всегда начинаются с `clang -cc1 -analyzer`; то есть, если требуется получить список всех команд анализатора, выполните команду:

```
$ clang -cc1 -help | grep analyzer
```

Использование статического анализатора в Xcode IDE

Пользующиеся интегрированной средой разработки Apple Xcode IDE могут использовать статический анализатор не покидая удобной среды. Для этого нужно сначала открыть проект и затем выбрать пункт **Analyze** (Анализ) в меню **Product** (Продукт). Вы увидите, что

статический анализатор Clang точно определяет место ошибки и позволяет среде разработки выделить ее, как показано на рис. 9.5.

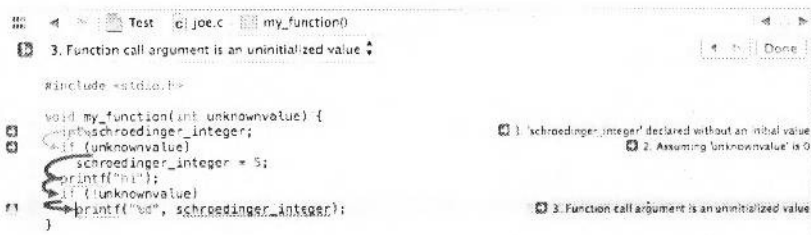


Рис. 9.5. Статический анализатор Clang точно определяет место ошибки

Анализатор может экспортировать информацию, если его вызвать с командой `plist`. Эта информация затем анализируется интегрированной средой разработки Xcode и выводится на экран в дружелюбном представлении.

Создание графических отчетов в формате HTML

Статический анализатор может также экспортировать информацию о подозрительных местах в программе в графическом виде – в формате HTML – подобно тому, как это делает Xcode. Для этого нужно передать анализатору параметр `-o` с именем каталога, куда следует сохранить файл отчета. Например:

```
$ clang -ccl -analyze -analyzer-checker=core joe.c -o report
```

При необходимости можно воспользоваться драйвером:

```
$ clang --analyze -Xanalyzer -analyzer-checker=core joe.c -o report
```

По этой команде анализатор обработает файл `joe.c` и сгенерирует отчет, похожий на изображение, что мы видели в окне Xcode, в виде файла HTML в каталоге `report`. Когда команда завершится, перейдите в каталог и откройте файл HTML. Вы должны увидеть отчет, как показано на рис. 9.6.

Анализ больших проектов

Если понадобится проверить статическим анализатором большой проект, вам едва ли захочется писать `makefile` или сценарий на `bash`,

чтобы вызвать анализатор для каждого файла с исходным кодом. В состав статического анализатора входит удобный инструмент – `scan-build` – как раз предназначенный для этой цели.

```

1
2 #include <stdio.h>
3
4 void my_function(int unknownvalue) {
5     int schroedinger_integer;
6     if (unknownvalue)
7         schroedinger_integer = 5;
8     printf("hi");
9     if (!unknownvalue)
10        printf("%d", schroedinger_integer);
11 }

```

1 'schroedinger_integer' declared without an initial value →

2 ← Assuming 'unknownvalue' is 0 →

3 ← Taking false branch →

4 ← Taking true branch →

5 ← Function call argument is an uninitialized value

Рис. 9.6. Отчет в формате HTML о проверке файла `joe.c`

Инструмент `scan-build` подменяет на время работы переменные окружения `CC` и `CXX`, определяющие команду вызова компилятора C/C++, и таким способом внедряет себя в процесс сборки проекта. Он анализирует каждый файл перед его компиляцией и по завершении позволяет системе сборки или сценарию продолжить работу в обычном режиме. По результатам анализа он формирует отчеты в формате HTML, которые можно просматривать в браузере. Структура команды очень проста:

```
$ scan-build <ваша команда сборки>
```

После `scan-build` вы можете указывать какие угодно команды сборки. Чтобы собрать программу Джо, например, обязательно ис-

пользовать Makefile – можно использовать прямую команду компиляции:

```
$ scan-build gcc -c joe.c -o joe.o
```

Когда она завершится, вы сможете выполнить команду scan-view, чтобы ознакомиться с содержимым отчета:

```
$ scan-view <каталог, куда scan-build сохранила отчеты>
```

Последняя строка, которую выведет scan-build, содержит значенные параметра для команды scan-view – путь к временному каталогу, где хранятся все сгенерированные отчеты. Открыв отчеты, вы должны увидеть хорошо отформатированные веб-страницы с сообщениями об ошибках для каждого файла с исходным кодом, как показано на рис. 9.7.

Bug Summary

Bug Type	Quantity	Display?
All Bugs	1	<input checked="" type="checkbox"/>
Logic error		
Uninitialized argument value	1	<input checked="" type="checkbox"/>

Reports

Bug Group	Bug Type -	File	Line	Path Length	
Logic error	Uninitialized argument value	joe.c	10	5	View Report Report Bug Open File

Рис. 9.7. Веб-страницы с сообщениями об ошибках для каждого файла с исходным кодом

Практический пример – поиск ошибок в исходном коде Apache

В этом примере мы покажем вам, насколько просто выполнить проверку большого (очень большого) проекта. Для этого откройте в браузере страницу <http://httpd.apache.org/download.cgi> и загрузите тарболл с исходным кодом последней версии Apache HTTP Server. На момент написания этих строк последней была версия 2.4.9. Мы выполнили загрузку в консоли и тут же распаковали полученный архив в текущий каталог:

```
$ wget http://archive.apache.org/dist/httpd/httpd-2.4.9.tar.bz2
$ tar -xjvf httpd-2.4.9.tar.bz2
```

Исследование исходного кода мы выполнили с помощью scan-build. Для этого нам пришлось сгенерировать сценарии сборки.

Имейте в виду, что вам потребуется установить все зависимости, необходимые для компиляции проекта Apache. После проверки и установки всех зависимостей нам осталось только выполнить следующую последовательность команд:

```
$ mkdir obj
$ cd obj
$ scan-build ../httpd-2.4.9/configure -prefix=$(pwd)/../install
```

Мы использовали параметр `-prefix`, чтобы указать новый путь к каталогу установки для этого проекта и избежать необходимости приобретения привилегий администратора. Однако, если вы не собираетесь устанавливать Apache, вам не потребуется указывать дополнительные параметры, при условии, что вы не будете выполнять команду `make install`. В данном случае мы указали путь установки, как имя каталога, находящегося в том же каталоге, куда были загружены и распакованы исходные тексты. Отметьте так же, что команда сборки начинается с вызова команды `scan-build`, которая переопределяет переменные окружения `CC` и `CXX`.

После того, как сценарий `configure` создаст все файлы `Makefile`, можно запускать фактический процесс сборки. Но, вместо простой команды `make`, мы выполнили команду:

```
$ scan-build make
```

Так как проект Apache не просто большой, а очень большой, анализ на нашем компьютере выполнялся несколько минут, и в результате было обнаружено 82 ошибки. На рис. 9.8 показано, как выглядит отчет.

После того, как позорная ошибка поразила все реализации OpenSSL и того внимания, которое она привлекла к себе, весьма любопытно наблюдать, как статический анализатор обнаруживает шесть вероятных ошибок в реализации Apache SSL, в файлах `modules/ssl/ssl_util.c` и `modules/ssl/ssl_engine_config.c`. Имейте в виду, что ошибочный код может никогда не выполняться на практике и в действительности его нельзя назвать ошибочным, просто анализатор ограничивает свою область видимости, чтобы закончить анализ в более или менее приемлемый промежуток времени. То есть, мы не утверждаем, что это действительно ошибки. Мы лишь показали, что статический анализатор тоже может ошибаться (рис. 9.9).

В этом примере мы видим, что статический анализатор показывает нам путь выполнения, который завершается присваиванием неопределенного значения переменной-члену `dc->nVerifyClient`. Часть

Bug Summary

Bug Type	Quantity	Display?
All Bugs	82	<input checked="" type="checkbox"/>
Dead store		
Dead assignment	12	<input checked="" type="checkbox"/>
Dead initialization	1	<input checked="" type="checkbox"/>
Logic error		
Assigned value is garbage or undefined	5	<input checked="" type="checkbox"/>
Branch condition evaluates to a garbage value	1	<input checked="" type="checkbox"/>
Called function pointer is null (null dereference)	1	<input checked="" type="checkbox"/>
Dereference of null pointer	20	<input checked="" type="checkbox"/>
Dereference of undefined pointer value	18	<input checked="" type="checkbox"/>
Division by zero	1	<input checked="" type="checkbox"/>
Result of operation is garbage or undefined	3	<input checked="" type="checkbox"/>
Uninitialized argument value	12	<input checked="" type="checkbox"/>
Unix API	7	<input checked="" type="checkbox"/>
Memory Error		
Memory leak	1	<input checked="" type="checkbox"/>

Рис. 9.8. Отчет об ошибках, полученный в результате проверки исходных текстов Apache

этого пути лежит через вызов функции `ssl_cmd_verify_parse()` – анализатор способен проследивать такие межпроцедурные пути выполнения, если функции находятся в том же модуле. В этой вспомогательной функции статический анализатор обнаружил ветку, в которой параметр `mode` не получает никакого значения и, соответственно, остается неинициализированным.

Примечание. Причина, по которой эта подозрительная ситуация в действительности не является ошибкой, заключается в том, что код в `ssl_cmd_verify_parse()` может обрабатывать все возможные значения `cmd_parms`, что действительно имеет место в этой программе (отметьте эту контекстную зависимость), и корректно инициализировать `mode` во всех этих случаях. Инструмент `scan-build`

обнаружил, что данный модуль, сам по себе, может выполняться по ошибочному пути, но у нас нет доказательств, что пользователи этого модуля передают неверные входные параметры. Статический анализатор не обладает достаточной мощностью, чтобы анализировать модуль в контексте целого проекта, потому что такой анализ потребовал бы неоправданно большого времени (вспомните об экспоненциальной зависимости алгоритма).

```

996 const char *ssl_cmd_SSLVerifyClient(cmd_parms *cmd,
997                                     void *defg,
998                                     const char *arg)
999 {
1000     SSLDirConfigRec *dc = (SSLDireConfigRec *)defg;
1001     SSLSrvConfigRec *sc = mySrvConfig(cmd->server);
1002     ssl_verify_t mode;
1003     const char *err;
1004
1005     if ((err = ssl_cmd_verify_parse(cmd, arg, &mode))) {
1006         return err;
1007     }
1008     if (cmd->path) {
1009         dc->nVerifyClient = mode;
1010

```

1 ← 'mode' declared without an initial value →

2 ← Calling 'ssl cmd verify parse' →

7 ← Returning from 'ssl cmd verify parse' →

8 ← Assuming 'err' is null →

9 ← Taking false branch →

10 ← Taking true branch →

11 ← Assigned value is garbage or undefined

Рис. 9.9. Подозрительные участки кода, которые фактически не являются ошибками

Этот конкретный путь состоит из 11 шагов, самый длинный путь, какой мы нашли в Apache, состоит из 42 шагов, находится в модуле

modules/generators/mod_cgid.c и нарушает требования стандарта C к вызову функций: он вызывает функцию `strlen()` с пустым указателем в аргументе.

Если вам любопытно увидеть все эти отчеты, не стесняйтесь, попробуйте запустить эти команду у себя.

Расширение статического анализатора Clang собственными средствами определения ошибок

Благодаря архитектуре анализатора, мы легко можем расширять его возможности, добавляя собственные средства проверки. Помните, что статический анализатор хорош ровно настолько, насколько хороши средства проверки, которыми он управляет, и если вам потребуется проверить, не использует ли какой-нибудь код ваш API непредусмотренным способом, вам придется изучить порядок добавления новых средств проверки в статический анализатор Clang.

Архитектура проекта

Исходные тексты статического анализатора находятся в каталоге `llvm/tools/clang`. Заголовочные файлы – в каталоге `include/clang/StaticAnalyzer`, а исходный код – в каталоге `lib/StaticAnalyzer`. Если заглянуть туда, можно заметить, что проект разбит на три разных подкаталога: `Checkers`, `Core` и `Frontend`.

Код в каталоге `Core` принадлежит симулятору выполнения программ на уровне исходного кода, который реализует шаблон проектирования «Посетитель» для вызова зарегистрированных средств проверки в каждой точке программы (до или после наиболее важных инструкций). Например, если ваше средство проверки пытается выявить попытки повторно освобождения одной и той же области памяти, оно, соответственно, должно наблюдать за вызовами функций `malloc()` и `free()`, и генерировать сообщения об ошибках при обнаружении таких попыток.

Механизм символического выполнения не может в точности имитировать все значения в программе, которые будут получаться во время выполнения. Если вы просите пользователя ввести целое число, вы будете точно знать, что в данном случае оно равно, например, 5. Мощь механизма символического выполнения заключается в его

способности определять, что получается в результате и, для достижения своей благородной цели, он работает с символами (`SVAl`), а не с конкретными значениями. Символ может соответствовать любому целому или вещественному числу, или даже совершенно неизвестному значению. Чем больше информации об этом значении, тем лучше.

Ключом к пониманию реализации проекта являются три структуры данных: `ProgramState`, `ProgramPoint` и `ExplodedGraph`. Первая представляет текущий контекст выполнения с учетом текущего состояния. Например, при анализе кода Джо, в ней был бы сохранен комментарий, что данная переменная имеет значение 5. Вторая представляет конкретную точку в потоке управления программы до или после инструкции, например, после инструкции присваивания значения 5 целочисленной переменной. Последняя представляет весь граф достижимых состояний программы. Узлы в этом графе представляются кортежами структур `ProgramState` и `ProgramPoint`, то есть каждая точка в программе имеет определенное состояние. Например, точка после присваивания значения 5 целочисленной переменной имеет состояние, связывающее число 5 с этой переменной.

Как уже отмечалось в начале этой главы, `ExplodedGraph` (граф достижимых состояний) обеспечивает более широкие возможности, чем классический граф CFG. Обратите внимание, что две следующие друг за другом, но не вложенные инструкции `if`, которые образуют маленький граф CFG, могли бы создать четыре ветви в графе достижимых состояний – это называют «комбинаторным взрывом». Для экономии места этот граф подвергается свертке, в том смысле, что если требуется создать узел, представляющий ту же точку в программе и с тем же состоянием, что и некоторый другой узел, анализатор не создает новый узел, а повторно использует существующий, возможно образуя циклы. Чтобы реализовать такое поведение, `ExplodedNode` наследует суперкласс `llvm::FoldingSetNode` из библиотеки LLVM. Библиотека включает общий класс для таких ситуаций, потому что свертка широко используется для представления программ в процессе компиляции.

Общую архитектуру статического анализатора можно разделить на следующие части: механизм, осуществляющий симуляцию и управляющий другими компонентами; диспетчер состояний, обслуживающий объекты `ProgramState`; диспетчер ограничений, который выводит ограничения из `ProgramState` в процессе симуляции выполнения данной ветки в программе; и диспетчер хранения, обслуживающий модель хранения данных в программе.

Другим важным аспектом анализатора является симуляция поведения памяти во время выполнения каждой ветки в программе. Это весьма сложная задача, потому что в таких языках, как C и C++ существует масса способов доступа к одной и той же области памяти.

Анализатор реализует модель областей памяти, описанную в статье Жонгксинга Сю (Zhongxing Xu) и других авторов (см. ссылки в конце этой главы), которая различает состояния даже отдельных элементов массивов. Сю (Xu) с соавторами предложил иерархическую организацию областей памяти, согласно которой, например, элемент массива является подобластью массива, который в свою очередь является подобластью стека. Каждое левостороннее (lvalue) значение в C, то есть, каждая переменная или разыменованная ссылка, имеет соответствующую область, моделирующую участок памяти. Содержимое каждой области памяти моделируется с помощью привязок. Каждая привязка связывает символическое значение с областью памяти. Мы понимаем, что вывалили на ваши головы слишком много информации, которую трудно переварить вот так сразу, поэтому давайте воспользуемся самым лучшим способом усвоения новых знаний – практической разработкой программного кода.

Разработка собственного средства проверки

Представьте, что вы работаете со специфическим встраиваемым программным обеспечением, которое управляет атомным реактором и опирается в своей работе на две основных функции: `turnReactorOn()` и `SCRAM()` (выключает реактор). Атомный реактор имеет активную зону с топливом, где протекает реакция распада, и управляющие стержни из материала, поглощающего нейтроны, которые замедляют реакцию и превращают атомную бомбу в атомную электростанцию.

Согласно условиям эксплуатации, двукратный вызов `SCRAM()` может привести к заклиниванию управляющих стержней, а двукратный вызов `turnReactorOn()` может вызвать неконтролируемое нарастание реакции. То есть, порядок использования функций строго регламентирован и ваша задача – исследовать огромный объем кода перед передачей его в эксплуатацию, чтобы убедиться, что он никогда не нарушит следующие правила:

- нигде в программе функция `SCRAM()` не вызывается два раза подряд – только через промежуточный вызов `turnReactorOn()`;

- нигде в программе функция `turnReactorOn()` не вызывается два раза подряд – только через промежуточный вызов `SCRAM()`.

В качестве примера возьмем следующий код:

```
int SCRAM();
int turnReactorOn();

void test_loop(int wrongTemperature, int restart) {
    turnReactorOn();
    if (wrongTemperature) {
        SCRAM();
    }
    if (restart) {
        SCRAM();
    }

    turnReactorOn();
    // код, поддерживающий работу реактора
    SCRAM();
}
```

Этот код нарушает требования использования API, если обе переменные – `wrongTemperature` и `restart` – имеют значения, отличные от нуля, потому что в этом случае функция `SCRAM()` вызывается два раза подряд, без промежуточного вызова `turnReactorOn()`. Он также нарушает требования, если оба параметра равны нулю, потому что в этом случае дважды вызывается функция `turnReactorOn()`.

Решение проблемы с помощью собственного средства проверки

Можно попробовать визуально проверить код, что весьма утомительно и есть риск ошибиться, а можно воспользоваться инструментом автоматической проверки, таким как статический анализатор Clang. Проблема в том, что статический анализатор ничего не знает о Nuclear Power Plant API. Мы можем исправить этот недостаток, реализовав специализированное средство проверки.

Сначала нужно определить понятия для модели состояний, касающиеся информации, которая должна распространяться через различные состояния программы. В данном случае нас интересует состояние реактора: включен или выключен. Мы можем не знать, включен реактор или нет; поэтому наша модель состояний должна включать три состояния: «неизвестно», «включено» и «выключено».

Теперь можно описать, как должно работать наше средство проверки.

Класс состояния

Перейдем к практической реализации. Возьмем за основу пример простого средства проверки в файле `SimpleStreamChecker.cpp`, находящегося в дереве каталогов Clang.

В каталоге `lib/StaticAnalyzer/Checkers` создадим новый файл `ReactorChecker.cpp` и начнем с определения класса, представляющего интересующее нас состояние:

```
#include "ClangSACheckers.h"
#include "clang/StaticAnalyzer/Core/BugReporter/BugType.h"
#include "clang/StaticAnalyzer/Core/Checker.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CallEvent.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"

using namespace clang;
using namespace ento;

class ReactorState {
private:
    enum Kind {On, Off} K;
public:
    ReactorState(unsigned InK): K((Kind) InK) {}
    bool isOn() const { return K == On; }
    bool isOff() const { return K == Off; }
    static unsigned getOn() { return (unsigned) On; }
    static unsigned getOff() { return (unsigned) Off; }
    bool operator==(const ReactorState &X) const {
        return K == X.K;
    }
    void Profile(llvm::FoldingSetNodeID &ID) const {
        ID.AddInteger(K);
    }
};
```

Наш класс содержит единственное поле данных `Kind`. Обратите внимание, что управление информацией о состоянии будет осуществляться классом `ProgramState`.

Неизменяемость экземпляров класса `ProgramState`

Класс `ProgramState` имеет одну интересную особенность – его экземпляры не изменяются. После создания, экземпляр этого класса никогда не изменяется: он хранит состояние, вычисленное для данной точки в данной ветке программы. В отличие от анализа потока данных, в ходе которого строится граф CFG, в данном случае мы имеем дело с графом достижимых состояний программы, где для каждой пары из точки в программе и ее состояния создается отдельный узел.

При таком подходе, если в программе имеется цикл, механизм анализа будет создавать новые ветки для каждой итерации. В анализе потоков данных, напротив, состояние тела цикла обновляется с каждой новой итерацией, пока не будет выполнен выход из цикла.

Однако, как подчеркивалось выше, когда механизм символического выполнения попадает в узел, представляющий ту же точку в теле цикла с тем же состоянием, он делает вывод об отсутствии в этой ветке новой информации, требующей обработки, и повторно использует уже имеющийся узел. С другой стороны, если состояние в теле цикла постоянно меняется, легко может быть достигнут предел: механизм прервет выполнение цикла после предопределенного числа итераций, которое можно задать при запуске инструмента.

Описание реализации

Так как после создания экземпляра, представляющий состояние, не изменяется, нам не требуется добавлять в класс `ReactorState` методы изменения свойств (`setters`) или функции-члены, изменяющие состояние экземпляра, но нам нужны конструкторы. Поэтому мы добавили конструктор `ReactorState(unsigned InK)`, принимающий целое число, представляющее текущее состояние реактора.

Наконец, функция `Profile` является следствием того факта, что `ExplodedNode` наследует `FoldingSetNode`. Такие методы должны предоставляться всеми подклассами, чтобы помочь механизму свертки LLVM отслеживать состояние узла и определять равенство двух узлов (и, соответственно, необходимость свертки). Соответственно, наша функция `Profile` указывает, что состояние узла отражает целое число `K`.

Вы можете использовать любые функции-члены класса `FoldingSetNodeID`, имена которых начинаются с `Add`, чтобы передать поля, уникально идентифицирующие этот экземпляр (см. `llvm/ADT/FoldingSet.h`). В данном случае мы использовали метод `AddInteger()`.

Определение подкласса Checker

Теперь объявим подкласс класса `Checker`:

```
class ReactorChecker : public Checker<check::PostCall> {
    mutable IdentifierInfo *IITurnReactorOn, *IISCRAM;
    OwningPtr<BugType> DoubleSCRAMBugType;
    OwningPtr<BugType> DoubleONBugType;
    void initIdentifierInfo(ASTContext &Ctx) const;
    void reportDoubleSCRAM(const CallEvent &Call,
```



```
CheckerContext &C) const;  
void reportDoubleON(const CallEvent &Call,  
                    CheckerContext &C) const;  
public:  
ReactorChecker();  
/// Обрабатывает turnReactorOn и SCRAM  
void checkPostCall(const CallEvent &Call, CheckerContext &C) const;  
};
```

Примечание. Начиная с версии Clang 3.5, вместо шаблона `OwningPtr<>` рекомендуется использовать стандартный шаблон `std::unique_ptr<>`. Оба шаблона представляют реализации интеллектуальных указателей (*smart pointer*).

Первые строки в нашем классе указывают, что он наследует класс `Checker` с параметром-шаблоном. В определениях подобных классов можно указывать множество параметров-шаблонов, соответствующих точкам в программе, которые должны проверяться нашим средством проверки. Технически параметры-шаблоны используются, чтобы породить нестандартный класс `Checker`, наследующий все классы, перечисленные в виде параметров. В данном случае наш класс наследует `PostCall`. Это наследование используется для реализации шаблона проектирования «Посетитель», когда наше средство проверки будет вызываться только для точек, представляющих интерес, и, как следствие этого, наш класс должен реализовать метод `checkPostCall`.

Возможно вас заинтересует возможность зарегистрировать свой класс для проверки разных типов точек в программе (см. `Checker-Documentation.cpp`). В данном примере интерес для нас представляют только точки, следующие непосредственно за вызовами функций, потому что нам требуется зафиксировать изменение состояния после вызова одной из функций управления реактором.

Ключевое слово `const` в объявлениях этих функций-членов отражает тот факт, что средства проверки не имеют информации о своем состоянии. Однако, нам необходимо кэшировать результаты извлечения объектов `IdentifierInfo`, представляющих символы `turnReactorOn()` и `SCRAM()`. Поэтому мы использовали ключевое слово `mutable`, чтобы обойти ограничения, накладываемые ключевым словом `const`.

Совет. Используйте ключевое слово `mutable` с большой осторожностью. В данном случае мы не вредим первоначальной идее организации средств проверки, потому что всего лишь кэшируем результаты, чтобы ускорить вычисления после второго вызова нашей реализации проверки, которая концептуально все так же остается

средством проверки без сохранения своего состояния. Ключевое слово `mutable` следует использовать только для поддержки мьютексов или таких вот сценариев кэширования.

Мы также должны сообщить инфраструктуре Clang, что реализовали обработку ошибки нового типа. Для этого следует создать экземпляры `BugType`, по одному для каждой новой ошибки: когда два раза подряд вызывается функция `SCRAM()` и когда два раза подряд вызывается функция `turnReactorOn()`. Мы также использовали класс `OwningPtr` для обертывания нашего объекта, который просто реализует автоматический указатель, автоматически освобождая память при уничтожении объекта `ReactorChecker`.

Только что объявленные классы – `ReactorState` и `ReactorChecker` – следует заключить в анонимное пространство имен. Это поможет компоновщику избежать необходимости экспортировать две наших структуры данных, которые, как мы знаем, будут использоваться только локально.

Макрос регистрации

Прежде чем погружаться в исследование реализации класса, нам нужно вызвать макрос, чтобы развернуть экземпляр `ProgramState`, используемый механизмом анализа, содержащий нашу информацию о состоянии:

```
REGISTER_MAP_WITH_PROGRAMSTATE(RS, int, ReactorState)
```

Обратите внимание, что вызов этого макроса записывается без точки с запятой в конце. Он устанавливает новую ассоциативную связь с каждым экземпляром `ProgramState`. Первый параметр может быть любым именем, которое позднее будет использоваться для ссылки на эти данные, второй параметр – тип ключа, и третий – тип сохраняемого объекта (в данном случае, класс `ReactorState`).

Средства проверки обычно используют ассоциативные связи для сохранения своего состояния, потому что связывание нового состояния с определенным ресурсом является распространенной практикой. Например, состояние «инициализировано» или «неинициализировано» каждой переменной, как описывалось в начале главы. В данном случае ключом ассоциативной связи является имя переменной, а хранимым значением – экземпляр класса, моделирующего состояние «инициализировано» или «неинициализировано». За дополнительной информацией о способах регистрации информации обращайтесь к определениям макросов в `CheckerContext.h`.

Имейте в виду, что в действительности необязательно устанавливать ассоциативные связи, потому что для каждой точки в программе мы всегда будем сохранять только одно состояние. Поэтому мы всегда будем использовать ключ 1 для доступа к хранимому значению.

Реализация подкласса Checker

Ниже приводится реализация конструктора нашего класса, осуществляющего проверку:

```
ReactorChecker::ReactorChecker() : IITurnReactorOn(0), IISCRAM(0) {
    // Инициализировать типы ошибок.
    DoubleSCRAMBugType.reset(new BugType("Double SCRAM",
                                          "Nuclear Reactor API Error"));
    DoubleONBugType.reset(new BugType("Double ON",
                                       "Nuclear Reactor API Error"));
}
```

Примечание. Начиная с версии Clang 3.5, вызов конструктора BugType должен выглядеть так:

```
BugType(this, "Double SCRAM", "Nuclear Reactor API Error")
и
BugType(this, "Double ON", "Nuclear Reactor API Error")
с передачей ссылки this в первом параметре.
```

Конструктор создает новые экземпляры класса BugType с описаниями новых типов ошибок, и передает их в вызов метода reset() класса OwningPtr. Здесь же инициализируются указатели IdentifierInfo.

Теперь определим вспомогательную функцию кэширования результатов обращения по этим указателям:

```
void ReactorChecker::initIdentifierInfo(ASTContext &Ctx) const {
    if (IITurnReactorOn)
        return;
    IITurnReactorOn = &Ctx.Idents.get("turnReactorOn");
    IISCRAM = &Ctx.Idents.get("SCRAM");
}
```

Объект ASTContext хранит определенные узлы AST (Abstract Syntax Tree – абстрактного синтаксического дерева) с типами и объявлениями, используемыми в пользовательской программе, и мы можем найти в нем и извлечь идентификатор функции.

Теперь реализуем функцию checkPostCall, реализующую шаблон проектирования «Посетитель». Не забывайте, что это const-функция – она не должна изменять состояние средства проверки:

```

void ReactorChecker::checkPostCall(const CallEvent &Call,
                                   CheckerContext &C) const {
    initIdentifierInfo(C.getASTContext());
    if (!Call.isGlobalCFunction())
        return;
    if (Call.getCalleeIdentifier() == IITurnReactorOn) {
        ProgramStateRef State = C.getState();
        const ReactorState *S = State->get<RS>(1);
        if (S && S->isOn()) {
            reportDoubleON(Call, C);
            return;
        }
        State = State->set<RS>(1, ReactorState::getOn());
        C.addTransition(State);
        return;
    }
    if (Call.getCalleeIdentifier() == IISCRAM) {
        ProgramStateRef State = C.getState();
        const ReactorState *S = State->get<RS>(1);
        if (S && S->isOff()) {
            reportDoubleSCRAM(Call, C);
            return;
        }
        State = State->set<RS>(1, ReactorState::getOff());
        C.addTransition(State);
        return;
    }
}

```

Так как при объявлении средства проверки мы указали, что оно должно выполняться после вызовов функций в программе, в первом параметре, типа `CallEvent`, передается информация о функции, вызванной непосредственно перед данной точкой (см. `CallEvent.h`). Второй параметр, типа `CheckerContext`, является единственным источником информации о текущем состоянии в этой точке программы, поскольку средство проверки, как мы знаем, не имеет собственного состояния. С помощью этого параметра мы получаем `ASTContext` и инициализируем объекты `IdentifierInfo`, необходимые для проверки вызовов интересующих нас функций. Далее с помощью объекта `CallEvent` определяется – была ли вызвана функция `turnReactorOn()`. Если это так, обрабатывается переход в состояние «включено».

Но перед этим проверяется, не было ли прежде установлено состояние «включено», что говорит о наличии ошибки в программе. Обратите внимание, что `RS` в инструкции `State->get<RS>(1)` – это просто имя, которое мы дали, когда регистрировали новое состояние

программы, а 1 – это фиксированное целое число, всегда обеспечивающее доступ к ассоциативной связи. Хотя в данном примере можно было обойтись без ассоциативной связи, ее применение может упростить расширение средства проверки в будущем и добавление в него новых, более сложных состояний.

Хранимое состояние извлекается как константный указатель, потому что мы имеем дело с информацией, достигшей данной точки программы, которая не может изменяться. Сначала нужно проверить, не был ли получен пустой указатель, соответствующий случаю, когда прежнее состояние реактора неизвестно. Если указатель непустой, далее проверяется – представляет ли он состояние «включено» и в случае положительного результата проверки мы прерываем дальнейший анализ и сообщаем об ошибке. В противном случае создается новое состояние вызовом метода `set` класса `ProgramStateRef`, которое затем передается методу `addTransition()` для записи и создания нового ребра в `ExplodedGraph`. Ребра создаются, только при фактическом изменении состояния. Аналогичная логика используется при исследовании вызова функции `SCRAM`.

Ниже показано, как мы реализовали функции вывода сообщений об ошибках:

```
void ReactorChecker::reportDoubleON(const CallEvent &Call,
                                     CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
        return;
    BugReport *R = new BugReport(*DoubleONBugType,
                                  "Turned on the reactor two times", ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
}

void ReactorChecker::reportDoubleSCRAM(const CallEvent &Call,
                                       CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
        return;
    BugReport *R = new BugReport(*DoubleSCRAMBugType,
                                  "Called a SCRAM procedure twice", ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
}
```

Прежде всего функции генерируют узел стока (`sink node`), который, в графе достижимых состояний программы, обозначает критическую

ошибку и сообщает, что нет смысла продолжать анализировать данную ветвь в программе. Следующие строки создают объект `BugReport`, указывающий, что найдена новая ошибка типа `DoubleOnBugType`. В конструктор объекта передаются тип ошибки, описание в свободной форме и только что созданный узел стока. Далее вызывается метод `addRange()`, который сохранит местоположение найденной ошибки.

Функция регистрации

Чтобы статический анализатор смог использовать наше средство проверки, нужно определить функцию регистрацию и добавить описание в файл `TableGen`. Ниже показано, как выглядит реализация функции регистрации:

```
void ento::registerReactorChecker(CheckerManager &mgr) {
    mgr.registerChecker<ReactorChecker>();
}
```

Файл `TableGen` (`lib/StaticAnalyzer/Checkers/Checkers.td`) содержит таблицу со списком средств проверки. Перед редактированием этого файла нужно решить, в каком пакете будет находиться наше средство проверки. Мы решили поместить его в пакет `alpha.powerplant`, а так как такого пакета пока нет, мы создали его. Откройте файл `Checkers.td` и добавьте новое определение в конец списка пакетов:

```
def PowerPlantAlpha : Package<"powerplant">, InPackage<Alpha>;
```

Затем добавьте определение нового средства проверки:

```
let ParentPackage = PowerPlantAlpha in {
    def ReactorChecker : Checker<"ReactorChecker">,
        HelpText<"Check for misuses of the nuclear power plant API">,
        DescFile<"ReactorChecker.cpp">;
} // конец "alpha.powerplant"
```

Если вы выполняли сборку Clang с помощью CMake, добавьте имя нового файла с исходным кодом в список `lib/StaticAnalyzer/Checkers/CMakeLists.txt`. Если вы пользовались сценарием `configure`, вам не нужно изменять никакие другие файлы, потому что утилита `make` сама обнаружит новый файл с исходным кодом в каталоге `Checkers` и добавит его в библиотеку средств проверки.

Сборка и тестирование

Перейдите в каталог, где выполнялась сборка LLVM и Clang, и выполните команду `make`. Система сборки обнаружит новый файл с ис-

ходным кодом, скомпилирует его и скомпирует со статическим анализатором Clang. По завершении сборки выполните команду `clang -cc1 -analyzer-checker-help` – она должна вывести список всех средств проверки, в том числе и вновь созданное.

Сохраните исходный код для тестирования в файле `managereactor.c` (это тот же код, что был представлен выше):

```
int SCRAM();
int turnReactorOn();

void test_loop(int wrongTemperature, int restart) {
    turnReactorOn();
    if (wrongTemperature) {
        SCRAM();
    }
    if (restart) {
        SCRAM();
    }
    turnReactorOn();
    // код, поддерживающий работу реактора
    SCRAM();
}
```

Чтобы проверить его с помощью нового средства проверки, выполните следующую команду:

```
$ clang --analyze -Xanalyzer -analyzer-checker=alpha.powerplant managereactor.c
```

В результате на экране должны появиться пути, в которых обнаружены ошибки. Если запросить сохранение результатов в формате HTML, вы получите отчет, как показано на рис. 9.10.

На этом ваше задание заканчивается: вы справились с созданием средства автоматической проверки соблюдения заданных требований. Если хотите, можете заглянуть в реализацию других средств проверки, чтобы больше узнать об особенностях обработки более сложных сценариев, или обратиться за дополнительной информацией к ресурсам, перечисленным в следующем разделе.

Дополнительные ресурсы

Ниже перечислены ресурсы, где можно найти дополнительную информацию и примеры проектов:

- <http://clang-analyzer.llvm.org>: страница проекта статического анализатора Clang;


```

1  int SCRAM();
2  int turnReactorOn();
3
4  void test_loop(int wrongTemperature, int restart) {
5      turnReactorOn();
6      if (wrongTemperature) {
7          SCRAM();
8      }
9      if (restart) {
10         SCRAM();
11     }
12     turnReactorOn();
13     // code to keep the reactor working
14     SCRAM();
15 }
16

```

1 Assuming 'wrongTemperature' is 0 →

2 ← Taking false branch →

3 ← Assuming 'restart' is 0 →

4 ← Taking false branch →

5 ← Turned on the reactor two times

Рис. 9.10. Отчет с результатами проверки тестовой программы `managereactor.c`

- http://clang-analyzer.llvm.org/checker_dev_manual.html: очень хорошее руководство с дополнительной информацией для тех, кто хочет заняться разработкой новых средств проверки;
- <http://lcs.ios.ac.cn/~xzx/memmodel.pdf>: статья «Memory Model for Static Analysis of C», авторы: Жонгксинг Сю (Zhongxing Xu), Тед Кременек (Ted Kremenek) и Цзянь Чжан (Jian Zhang). Подробно описывает теоретические аспекты модели памяти, которая была реализована в ядре анализатора;
- <http://clang.llvm.org/doxygen/annotated.html>: документация с описанием Clang;

- <http://llvm.org/devmtg/2012-11/videos/Zaks-Rose-Checker-24Hours.mp4>: лекция на встрече разработчиков LLVM в 2012 году, где Анна Закс (Anna Zaks) и Джордан Роуз (Jordan Rose), разработчики статического анализатора, рассказывают, как быстро создать средство проверки.

В заключение

В этой главе мы узнали, чем статический анализатор Clang отличается от простых инструментов поиска ошибок, которые используются компиляторами. Мы показали примеры ситуаций, где статический анализатор оказывается более точным в своих оценках, и объяснили, что высокая точность достигается ценой более существенных затрат времени, что алгоритм работы статического анализатора имеет экспоненциальную временную сложность и именно поэтому его нежелательно встраивать в конвейер компиляции. Мы также показали, как использовать интерфейс командной строки для запуска статического анализатора для проверки простых проектов, и познакомили вас со вспомогательным инструментом `scan-build`, упрощающим анализ больших проектов. В конце главы мы рассказали, как добавить в статический анализатор свое средство проверки.

В следующей главе мы поговорим об инструментах Clang, построенных на основе фреймворка LibTooling, который упрощает реализацию утилит рефакторинга кода.

ГЛАВА 10.

Инструменты Clang и фреймворк LibTooling

В этой главе мы увидим, как много инструментов используют анализатор исходных текстов Clang в качестве библиотеки для выполнения самых разных операций с исходным кодом на C/C++. В частности, все они опираются на фреймворк LibTooling, библиотеку Clang, которая может служить основой для создания автономных инструментов. В данном случае речь идет не о расширениях, встраиваемых в процесс компиляции, а об отдельных инструментах, пользующихся средствами парсинга Clang, что дает возможность использовать их непосредственно. Инструменты, представленные в этой главе, доступны в виде пакета Clang Extra Tools. Подробнее о том, как установить их, рассказывается в главе 2, «Внешние проекты». А в конце этой главы мы рассмотрим пример создания собственного инструмента рефакторинга. В этой главе охватываются следующие темы:

- ♦ создание базы данных команд компиляции;
- ♦ приемы использования некоторых инструментов Clang, опирающихся на LibTooling, таких как Clang Tidy, Clang Modernizer, Clang Apply Replacements, ClangFormat, Modularize, PPTrace и Clang Query;
- ♦ создание на основе LibTooling собственного инструмента рефакторинга исходного кода.

Создание базы данных команд КОМПИЛЯЦИИ

В общем случае компилятор вызывается из сценария сборки, например, Makefile, с последовательностью параметров, которые задают местоположение заголовочных файлов проекта и содержат некоторые

определения. Эти параметры позволяют анализатору исходного текста провести лексический и синтаксический анализ исходного кода. Однако, в этой главе мы будем знакомиться с самостоятельными инструментами, не встраиваемыми в процесс компиляции. То есть, теоретически, нам может потребоваться написать сценарий, чтобы обеспечить запуск инструментов с нужными параметрами для каждого исходного файла.

Например, ниже приводится полная команда, которая используется утилитой `make` для вызова компилятора с целью компиляции типичного файла из библиотеки LLVM:

```
$ /usr/bin/c++ -DNDEBUG -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS
-D__STDC_LIMIT_MACROS -fPIC -fvisibility-inlines-hidden -Wall -W -Wno-
unused-parameter -Wwrite-strings -Wmissing-field-initializers -pedantic
-Wno-long-long -Wcovered-switch-default -Wnon-virtual-dtor -fno-rtti
-I/Users/user/p/llvm/llvm-3.4/cmake-scripts/utils/TableGen -I/Users/
user/p/llvm/llvm-3.4/llvm/utils/TableGen -I/Users/user/p/llvm/llvm-3.4/
cmake-scripts/include -I/Users/user/p/llvm/llvm-3.4/llvm/include -fno-
exceptions -o CMakeFiles/llvm-tblgen.dir/DAGISelMatcher.cpp.o -c /Users/
user/p/llvm/llvm-3.4/llvm/utils/TableGen/DAGISelMatcher.cpp
```

Если придется работать с такой библиотекой, вам едва ли понравится набирать подобные команды длиной по 10 строк, чтобы проанализировать каждый исходный файл. И да, вы не сможете отказаться ни от одного символа, потому что анализатор исходного кода использует каждый бит этой информации.

Чтобы упростить обработку исходных файлов, любой проект, использующий LibTooling, принимает на входе базу данных. Эта база данных содержит все необходимые параметры компилятора для каждого исходного файла в данном проекте. Чтобы упростить эту задачу, CMake может генерировать такие базы данных, если указать ключ `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`. Например, предположим, что нам требуется воспользоваться инструментом на основе LibTooling для обработки исходного файла из проекта Apache. Чтобы избавиться от необходимости передавать полный комплект параметров компилятора для корректного анализа этого файла, с помощью CMake можно сгенерировать базу данных команд, как показано ниже:

```
$ cd httpd-2.4.9
$ mkdir obj
$ cd obj
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ../
$ ln -s $(pwd)/compile_commands.json ../
```

Чем-то напоминает процедуру сборки Apache с помощью CMake, только вместо фактической сборки, из-за наличия в командной строке флага `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`, CMake создаст файл в формате JSON с параметрами компилятора, которые могли бы использоваться для компиляции каждого исходного файла в проекте Apache. Далее нужно создать ссылку на этот файл в корневом каталоге дерева с исходными текстами Apache. Теперь, запуская любую программу на основе LibTooling для анализа исходных файлов из проекта Apache, она будет просматривать родительские каталоги, пока не найдет `compile_commands.json` с параметрами для парсинга исходного файла.

Как вариант, если нет желания генерировать базу данных команд компиляции перед вызовом своего инструмента, можно через двойной дефис (`--`) передать параметры компиляции непосредственно. Этот прием может пригодиться, если для сборки проекта не требуется передавать слишком много параметров. Например, взгляните на следующую команду:

```
$ my_libtooling_tool test.c -- -Икаталог_с_заголовочными_файлами  
-Днекоторое_определение
```

Clang-tidy

В этом разделе мы рассмотрим `clang-tidy`, как пример инструмента на основе LibTooling, и расскажем, как им пользоваться. Все остальные инструменты Clang используются похожим образом, поэтому вы легко сможете заняться их исследованием самостоятельно.

Инструмент `clang-tidy` проверяет наличие в исходном коде распространенных нарушений стандартов оформления. Он может проверять такие характеристики, как:

- переносимость кода между разными компиляторами;
- следование определенным идиомам и соглашениям;
- возможность появления ошибок из-за злоупотребления опасными особенностями языка.

В частности, `clang-tidy` может использовать два типа средств проверки: входящие в состав статического анализатора Clang и специально написанные для `clang-tidy`. Несмотря на возможность использовать проверки, выполняемые статическим анализатором, имейте в виду, что `clang-tidy` и другие инструменты на основе LibTooling анализируют исходный код, чем существенно отличаются

от механизма статического анализа, описанного в предыдущей главе. Вместо имитации выполнения программы, эти инструменты просто просматривают дерево AST и действуют намного быстрее. В отличие от средств проверки в составе статического анализатора Clang, проверки, написанные для clang-tidy, обычно нацелены на определение соответствия или несоответствия определенным соглашениям по оформлению исходного кода. В частности, они проверяют соответствие соглашениям, принятым в LLVM или в Google, а также соответствие некоторым другим параметрам.

Если вы следуете какому-то определенному соглашению, clang-tidy пригодится вам для периодической проверки своего кода. Приложив некоторые усилия, в отдельных редакторах можно даже настроить вызов этого инструмента. Однако в настоящее время инструмент пока находится на начальном этапе своего развития и реализует пока небольшое число проверок.

Проверка исходного кода с помощью Clang-tidy

В этом примере мы покажем, как с помощью clang-tidy проверить исходный код, написанный в главе 9, «Статический анализатор Clang». Там мы реализовали собственное расширение для статического анализатора, и если бы мы захотели передать его в официальное дерево исходных текстов Clang, нам потребовалось бы обеспечить строгое соответствие соглашениям об оформлении, принятым в LLVM. Итак, давайте проверим, насколько далеко мы отклонились от этих соглашений. В общем виде интерфейс командной строки clang-tidy выглядит, как показано ниже.

```
$ clang-tidy [параметры] <файл0> [... <файлN>] [-- <команда компилятора>]
```

Можно со всем тщанием указывать необходимые проверки в параметре `-checks`, а можно воспользоваться групповым оператором `*` и выбрать сразу множество проверок, названия которых начинаются с указанной подстроки. Если потребуется запретить какие-то проверки, достаточно просто добавить минус перед их именами. Например, чтобы запустить все проверки, имеющие отношение к соглашениям по оформлению исходных текстов в LLVM, можно выполнить следующую команду:

```
$ clang-tidy -checks="llvm-*" file.cpp
```

Совет. Все инструменты, описанные в этой главе, доступны, только если вместе с Clang устанавливались дополнительные инструменты из проекта *Clang Extra Tools*, который развивается отдельно. Если у вас еще не установлен инструмент `clang-tidy`, прочитайте главу 2, «Внешние проекты», где приводятся инструкции по сборке и установке *Clang Extra Tools*.

Так как наш код компилируется вместе с Clang, нам потребуется база данных команды компиляции. Поэтому начнем с ее создания. Перейдите в каталог, где находятся исходные тексты LLVM, и создайте отдельный каталог для хранения файлов CMake:

```
$ mkdir cmake-scripts
$ cd cmake-scripts
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ../llvm
```

Совет. Если вы столкнетесь с ошибкой, сообщающей о неизвестном исходном файле и ссылающейся на файл с реализацией нашего средства проверки, созданный в предыдущей главе, просто дайте имя этого файла в `CMakeLists.txt`. Для этого выполните следующую команду и затем запустите CMake еще раз:

```
$ vim ../llvm/tools/clang/lib/StaticAnalyzer/Checkers/
CMakeLists.txt
```

Затем в корневом каталоге LLVM создайте ссылку на файл базы данных команд компиляции.

```
$ ln -s $(pwd)/compile_commands.json ../llvm
```

Теперь можно вызвать сам инструмент `clang-tidy`:

```
$ cd ../llvm/tools/clang/lib/StaticAnalyzer/Checkers
$ clang-tidy -checks=>llvm-*> ReactorChecker.cpp
```

На экране должен появиться длинный список с жалобами на заголовочные файлы, подключаемые нашим средством проверки, которые не соответствуют правилу LLVM, требующему добавлять комментарий после каждой фигурной скобки, закрывающей пространство имен (см. <http://llvm.org/docs/CodingStandards.html#namespace-indentation>). Но есть и приятная новость: наш исходный код, кроме заголовочных файлов, не нарушает ни одного требования.

Инструменты рефакторинга

В этом разделе мы познакомим вас с многими другими инструментами, анализирующими исходный код и выполняющими его преоб-

разование, используя для этого возможности парсинга Clang. Вы без труда сможете использовать их, так как интерфейс командной строки этих инструментов ничем не отличается от интерфейса clang-tidy и опирается на использование базы данных.

Clang Modernizer

Clang Modernizer – это революционный инструмент, цель которого помочь пользователям адаптировать старый код на C++ под новейшие стандарты, такие как C++11. Выполняется такая адаптация путем выполнения следующих преобразований:

- **преобразование циклов:** циклы в старом стиле языка C `for (; ;)` преобразуются в более новые циклы вида: `for(auto &...:..);`
- **преобразование пустых указателей:** константы `NULL` и `0`, используемые для представления пустых указателей, замещаются новым ключевым словом `nullptr`, появившимся в стандарте C++11;
- **преобразование с добавлением ключевого слова `auto`:** в некоторые объявления включается ключевое слово `auto` для большей удобочитаемости кода;
- **преобразование с добавлением ключевого слова `override`:** добавляет в объявления методов, переопределяющих виртуальные методы родительских классов, спецификатор `override`;
- **преобразование передачи аргументов по значению:** передача константных ссылок замещается передачей по значению с выполнением операции копирования;
- **преобразование с заменой `auto_ptr`:** замещает устаревшие автоматические указатели `std::auto_ptr` современными `std::unique_ptr`.

Clang Modernizer – отличный пример инструментов, осуществляющих преобразование исходного кода, которое стало возможным благодаря фреймворку Clang LibTooling. Ниже приводится обобщенный синтаксис командной строки этого инструмента:

```
$ clang-modernize [параметры] <файл0> [... <файлN>] [-- <команда компилятора>]
```

Обратите внимание, что если не указывать никаких дополнительных параметров, кроме имен файлов с исходными текстами, все преобразования будут применены непосредственно к этим файлам. Используйте флаг `-serialize-replacements`, чтобы заставить ин-

струмент записывать предлагаемые преобразования, или поправки на диск отдельно и получить возможность ознакомиться с ними и применить их, если вас все устраивает. Для применения таких «заплат» существует специальный инструмент, о котором мы расскажем далее.

Clang Apply Replacements

Разработка Clang Modernizer (прежде, C++ Migrator) породила дискуссии о том, как лучше координировать преобразования в больших проектах. Например, при анализе разных единиц трансляции, могут многократно анализироваться одни и те же заголовочные файлы.

Одно из возможных решений заключается в том, чтобы сохранять предлагаемые поправки в файл. Второй инструмент, который мы рассмотрим, отвечает за чтение файлов с поправками, отсеивает противоречивые и повторяющиеся поправки, и применяет их к файлам с исходными текстами. Эту задачу решает Clang Apply Replacements, созданный в помощь Clang Modernizer для случаев, когда необходимо внести поправки в большие проекты.

Оба инструмента – Clang Modernizer, который производит поправки, и Clang Apply Replacements, который применяет поправки, – работают с сериализованной версией класса `clang::tooling::Replacement`. Сериализация выполняется в формат YAML, который можно определить, как надмножество формата JSON.

Файлы «заплат» (patch files), порождаемые и используемые инструментами ревизии кода, являются точным представлением сериализованных поправок, но разработчики Clang решили отдать предпочтение формату YAML для представления экземпляров `Replacement` и избежать необходимости выполнять парсинг файлов заплат.

По этой причине Clang Apply Replacements не является универсальным инструментом наложения «заплат» – это специализированный инструмент, ориентированный на применение поправок, созданных инструментами Clang на основе LibTooling. Обратите внимание, что если вы решите заняться разработкой инструмента преобразования исходного кода, инструмент Clang Apply Replacements потребуется вам только для координации наложения множества поправок с поддержкой удаления повторяющиеся исправлений. В противном случае исправления можно было бы накладывать на файл непосредственно.

Чтобы опробовать Clang Apply Replacements, нам нужно сначала воспользоваться инструментом Clang Modernizer и заставить его сохранить поправки в файл. Допустим, что нам захотелось преобразо-

вать следующий файл `test.cpp` с исходным кодом на C++, чтобы привести его в соответствие с требованиями новейших стандартов C++:

```
int main() {
    const int size = 5;
    int arr[] = {1,2,3,4,5};
    for (int i = 0; i < size; ++i) {
        arr[i] += 5;
    }
    return 0;
}
```

Согласно утверждениям в руководстве пользователя к Clang Modernizer, этот цикл можно безопасно преобразовать в цикл с использованием итератора `auto`. Для этого следует выполнить преобразование циклов:

```
$ clang-modernize -loop-convert -serialize-replacements test.cpp
--serialize-dir=./
```

Последний параметр можно опустить – он указывает, что файлы с поправками должны сохраняться в текущем каталоге. Если вызвать инструмент без этого параметра, он создаст временный каталог, который потом будет использован инструментом Clang Apply Replacements. Поскольку все преобразования сохраняются в текущем каталоге, вы легко сможете исследовать сгенерированные файлы YAML. Чтобы применить исправления, просто выполните команду `clang-apply-replacements`, передав ей путь к текущему каталогу:

```
$ clang-apply-replacements ./
```

Совет. Если в процессе выполнения этой команды появилось сообщение об ошибке «*trouble iterating over directory ./: too many levels of symbolic links*» (ошибка обхода каталога `./`: слишком много уровней вложенности символических ссылок), попробуйте выполнить две предыдущие команды с сохранением исправлений в каталоге `/tmp`. Как вариант, можете попробовать создать новый каталог для хранения этих файлов.

Инструменты, подобные описываемым здесь, обычно создаются с целью упростить обработку больших объемов исходного кода. Поэтому Clang Apply Replacements не задает никаких вопросов в процессе выполнения, а просто выполняет парсинг всех файлов YAML, доступных в указанном каталоге, анализирует их и применяет преобразования.

Вы можете даже определить стандарты, которым должен следовать инструмент при наложении исправлений. Для этой цели используется флаг `-style=<LLVM|Google|Chromium|Mozilla|Webkit>`. Данная функциональность, реализованная в библиотеке LibFormat, позволяет любым инструментам рефакторинга создавать новый исходный код в заданном формате и с соблюдением заданных соглашений. С этой замечательной особенностью мы познакомимся в следующем разделе.

ClangFormat

Представьте, что вы – судья соревнований, подобных международному конкурсу запутывания кода на Си (International Obfuscated C Code Contest, IOCCC). Чтобы дать вам возможность понять, в чем заключается этот конкурс, приведем программный код (рис. 10.1), написанный победителем в 20-секундной номинации, Майклом Биркеном (Michael Birken). Имейте в виду, что этот код распространяется на условиях лицензии Creative Commons Attribution-ShareAlike 3.0, которая допускает свободную модификацию кода, при условии, что вы поддерживаете лицензию и указываете принадлежность IOCCC.

Если кто-то из вас не верит – это действительно допустимый код на С. Вы можете загрузить его по адресу: <http://www.ioccc.org/2013/birken>. А теперь давайте продемонстрируем, что может сделать ClangFormat с этим примером:

```
$ clang-format -style=llvm obf.c --
```

Результат показан на рис. 10.2.

Лучше, не правда ли? К счастью, в реальной жизни нет необходимости иметь дело с запутанным кодом, как в этом примере, но исправление форматирования в соответствии с определенными соглашениями тоже нельзя назвать работой мечты. Эту рутинную работу берет на себя ClangFormat. ClangFormat – не только инструмент, но и библиотека LibFormat, которая реализует форматирование кода в соответствии с соглашениями. То есть, если вы решите заняться созданием инструмента, генерирующего программный код на С или С++, вы сможете переложить задачу форматирования на ClangFormat и сосредоточиться на своем проекте.

Помимо расстановки переносов строк и отступов ClangFormat может гораздо больше. Это весьма хитроумный инструмент, позволяющий разбить код на 80-символьные строки и улучшить его удобочи-

таемость. Если вам когда-либо приходилось останавливаться, чтобы подумать, как лучше разбить длинную инструкцию, вы оцените, насколько хорошо с этой задачей справляется ClangFormat. Настройте его использование в качестве внешнего инструмента в своем редакторе и назначьте горячую клавишу для его запуска. Если вы пользуетесь таким известным редактором, как Vim или Emacs, знайте, что нашлись люди, которые уже написали сценарии для интеграции в них инструмента ClangFormat.

```

c| obf.c
|char* = "****/";
#include <stdio.h>
#define m 21
#define o(l, k) for(l=0; l<k; l++)
#define n(k) o(T, k)

int E,L,O,R,G[42][m],h[2][42][m],g[3][0],c
[42][42][2],f[42]; char d[42]; void v( int
b,int a,int j){ printf("\33[%d;%d\33[4m"
"m ",a,b,j); } void u(){ int T,e; n(42)o(
e,m)if(h[0][T][e]-h[1][T][e]){ v(e+4,e,T-2
,h[0][T][e]+17*h[0][T][e]); h[1][T][e]=h[
0][T][e]; } fflush(stdout); } void q(int l
,int k,int p){
int T,c,a; L=0
; O=1; while(O
){ n(4&&L){ e=
k+c[l][T][0];
h[0][L-1+c[l][
T][1]][p720-e:
e]--; } n(4){
l++; if(a==42
; } } n(4){ e=
T][1]][p720-e:
u(); } n(42){
o(a, m&&e==m){
}h[0][L-1][a
]};int main(){ int ,e,t,r,i,s
l][T]=7-T; R--; n(42) o(e,m)
R; n(17){ e=d[T]-48; d[T]=0;
} } n(8)if(g[0][7-T]){ t=g[i]
g[2][g[i][T]]=T; n(R+1)o(e,m
)o(t,2){ f(T+t-T)-T["%s",9"
[e]t]="5"<s>8)2h-h918"9"
} } n(15) { s=t>9?m:(T&3)-3715:36;o(e,s)o(t,2)c[T+19][e][t]="*6,8*6.608.6264826668865:\
865:1;+;0(6+6-6/8)163638065678469.;88)1{f6*6,5.60626466666:0)8-0*818.E502/9863(+;/"*6,5.6)
0626466666:4)8-8*818.8582/9863(+;/,6.606264666666:8-018.8582/8064+4+4(0)-;/.6062/
6466666:8/8300/7044,4-4+4+4(0)163+;/0626466666:818582/9064,4/4,4-4+4+4(0)1+;" [e+e
+e+e]40; E+=s+s; } n(45){ if(T>l) { v(2,T,7); v(46,T,7); } v(7+T,44,7); T=0; o(e,
42)o(t,m)h[T][e]--; while(R+i) { s = 0=0; if (r-R) { c[L+19][T+i] V=7/2
; else if(G[R][T]+1) s++; if(s) { if(V>4){ V=9-V; D++; } V+=20; n(20) q(c[V][T][0],c
[V][T][1],D); } } n(19) if((L=G[R][T]+1) { 0=T-L; e=0+9; t=718-0 :0; o(K,(t&3)-3?
16:37){ if(K){ L=c[t+19][K-1][0]; 0=c[t+19][K-1][1]; } } } if(s) q(
c[V][20][0], c[V][20][1], D); R--; } printf("\33[47;1*\33[725h\33[40m"); return 0; }

```

Рис. 10.1. Программный код, победивший в 20-секундной номинации

```

c de-obf.c · No Selection
#include <stdio.h>
#define n 21
#define o(l, k) for (l = 0; l < k; l++)
#define n(k) o(T, k)

int E, L, O, R, G[42][m], h[2][42][m], g[3][0], c[42][42][2], f[42];
char o[42];

void v(int b, int a, int j) {
    printf("\33[33m\33[44m"
           "
           a, b, j);
}

void u() {
    int T, e;
    n(42) o(e, m) { h[0][T][e] - h[1][T][e] }
    v(e + 4 + e, T + 2, h[0][T][e] + 1 ? h[0][T][e] : 0);
    h[1][T][e] = h[0][T][e];
}
fflush(stdout);
}

void q(int l, int k, int p) {
    int T, e, a;
    L = 0;
    O = 1;
    while (0) {
        n(4 && L) {
            e = k + c[l][T][0];
            n[0][L - 1 + c[l][T][1]][p ? 20 - e : e] = -1;
        }
        n(4) {
            e = k + c[l][T][0];
            a = L + c[l][T][1] + 1;
            if (a == 42 || h[0][a][p ? 20 - e : e] + 1) {
                O = 0;
            }
        }
        n(4) {
            e = k + c[l][T][0];
            n[0][L + c[l][T][1]][p ? 20 - e : e] = g[1][f[p ? 19 + l : l]];
        }
        L++;
        u();
    }
}

```

Рис. 10.2. Результат преобразования запутанного кода с помощью ClangFormat

Тема форматирования кода, его организации и увеличения ясности плавно перетекает в проблему, характерную для языка C/C++: злоупотребление заголовочными файлами и координация их использования. Мы посвятили весь следующий раздел обсуждению одного из решений этой проблемы и применения инструментов Clang для его практического применения.

Modularize

Чтобы понять цель проекта Modularize, мы сначала познакомим вас с понятием модулей в языках C и C++, для чего отклонимся от главной

темы этой главы. Когда мы писали эти строки, модули еще не были официально стандартизованы. Читатели, которым не интересно узнать, как Clang реализует эту новую идею для проектов на C/C++, могут просто пропустить данный раздел и перейти к обсуждению следующего инструмента.

Программный интерфейс на C/C++

В настоящее время программы на C и C++ делятся на заголовочные файлы, например, файлы с расширением `.h`, и файлы реализации, например, файлы с расширением `.c` или `.cpp`. Компилятор интерпретирует каждую комбинацию файла реализации и подключаемых им заголовочных файлов, как отдельную единицу трансляции.

Программируя на C или C++ и работая с определенным файлом реализации, необходимо подумать о том, какие сущности должны принадлежать локальной области видимости, а какие – глобальной. Например, функции или элементы данных, которые не будут использоваться разными файлами реализации, должны объявляться в языке C с ключевым словом `static`, а в C++ – в анонимном пространстве имен. Эти признаки сообщают компоновщику, что данная единица трансляции не экспортирует локальные сущности и потому они не должны быть доступны другим единицам трансляции.

Однако, если та или иная сущность должна быть доступна сразу нескольким единицам трансляции, возникает проблема. Ради ясности будем называть экспортерами единицы трансляции, экспортирующие сущности, и импортерами, или пользователями – единицы трансляции, импортирующие эти сущности. Также предположим, что экспортер с именем `gamelogic.c` экспортирует простую целочисленную переменную `num_lives` импортеру с именем `screen.c`.

Задача компоновщика

Прежде всего посмотрим, как компоновщик обрабатывает импортируемый символ в нашем примере. В результате компиляции и ассемблирования `gamelogic.c` создается объектный файл `gamelogic.o`, таблица символов в которого сообщает, что символ `num_lives` имеет размер 4 байта и доступен для использования любыми другими единицами трансляции.

```
$ gcc -c gamelogic.c -o gamelogic.o
$ readelf -s gamelogic.o
```

Чис:	Знач	Разм	Тип	Связ	Vis	Индекс имени
7	00000000	4	ОБЪКТ	GLOBAL	DEFAULT	3 num_lives

В этой таблице мы оставили только интересующий нас символ. Инструмент `readelf` доступен только для платформ Linux, основанных на использовании широко распространенного формата **ELF (Executable and Linkable Format – формат выполняемых и компонуемых модулей)**. На других платформах таблицу символов можно вывести командой `objdump -t`. Эту таблицу следует читать так: наш символ `num_lives` находится в седьмой позиции в таблице и занимает первый адрес (ноль) относительно раздела с индексом 3 (раздел `.bss`). Раздел `.bss` хранит данные, которые инициализируются нулями. Чтобы проверить соответствие между именами разделов и их индексами, выведите заголовки разделов командой `readelf -S` или `objdump -h`. Из этой таблицы также следует, что символ `num_lives` является объектом (данными), имеющим размер 4 байта и глобальную область видимости (глобальную привязку).

Файл `screen.o` имеет аналогичную таблицу символов, которая сообщает, что данная единица трансляции импортирует символ `num_lives`, принадлежащий другой единице трансляции. Для получения таблицы выполним те же команды:

```
$ gcc -c screen.c -o screen.o
$ readelf -s screen.o
```

Чис:	Знач	Разм	Тип	Связ	Vis	Индекс имени
10	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND num_lives

Элемент таблицы похож на тот, что мы видели в таблице символов экспортера, но содержит меньше информации. Здесь отсутствует информация о типе и размере, а поле индекса сообщает, что раздел ELF, где хранится этот символ, не определен (`UND – undefined`), что характеризует данную единицу трансляции как импортера. Если эту единицу трансляции включить в состав программы, компоновщик не сможет справиться со своей работой, если не сможет разрешить данную зависимость.

Компоновщик принимает оба файла и подставляет в импортера адрес и описание символа, находящегося в экспортере.

```
$ gcc screen.o gamelogic.o -o game
$ readelf -s game
```

Чис:	Знач	Разм	Тип	Связ	Vis	Индекс имени
60	0804a01c	4	ОБЪЕКТ	GLOBAL	DEFAULT	25 num_lives

Теперь в поле значения хранится полный адрес переменной в виртуальной памяти, куда загружается программа и который может использоваться в разделе кода импортера.

Как видите, совместное использование сущностей несколькими единицами трансляции на стороне компоновщика организуется просто и эффективно.

На стороне анализатора исходного кода

Простота обработки объектных файлов никак не отражается на самом языке. Анализируя реализацию импортера, компилятор, в отличие от компоновщика, не может положиться только на одно имя импортируемой сущности, потому что ему необходимо убедиться, что семантика данной единицы трансляции не нарушает систему типов языка; ему пужно точно знать, что `num_lives` имеет целочисленный тип. Поэтому компилятор ожидает получить информацию о типе вместе с именем импортируемой сущности. Исторически эта проблема в языке С решается с помощью заголовочных файлов.

В заголовочных файлах хранятся объявления типов вместе с именами сущностей, которые используются разными единицами трансляции. Согласно этой модели, импортер использует директиву `include`, чтобы загрузить информацию о типах импортируемых им сущностей. Однако, заголовочные файлы могут использоваться более гибким способом, чем необходимо, и способны нести в себе не только объявления, но и программный код на С или С++.

Проблема препроцессора С/С++

В отличие от директивы `import` в таких языках программирования, как Java, семантика директивы `include` не ограничивается передачей компилятору необходимой информации об импортируемых символах. Фактически она добавляет дополнительный код на С или С++, который необходимо проанализировать. Этот механизм реализует препроцессор, который слепо копирует и изменяет код перед фактической компиляцией, и он ничуть не интеллектуальнее других инструментов обработки текста.

Такое раздувание объема кода еще больше осложняется в языке С++, где применение шаблонов способствует к включению в заголовочные файлы объявлений целых классов, из-за чего увеличивается объем внедряемого кода во все импортеры, использующие эти заголовочные файлы.

Это обстоятельство ложится тяжелым бременем на компилятор, особенно в проектах на С или С++, опирающихся на множество библиотек (или внешних сущностей), потому что компилятору придется анализировать одни и те же заголовочные файлы по многу раз.

Примечание. Проблема импортирования и экспортирования сущностей, которая решается компоновщиком с помощью таблиц символов, требует от компилятора выполнить парсинг тысяч строк в заголовочных файлах, написанных человеком.

В больших проектах компиляторов обычно реализуется схема использования предварительно скомпилированных заголовочных файлов, чтобы избежать необходимости выполнять лексический анализ снова и снова. В Clang для этого создаются файлы PCH. Однако, этот прием всего лишь ослабляет проблему, но не решает ее, потому что компилятору все еще необходимо, например, интерпретировать заголовочный файл из-за возможности появления новых определений макросов, влияющих на то, как текущая единица трансляции видит этот заголовочный файл.

Например, представьте что в нашей программе используется следующий заголовочный файл `gamelogic.h`:

```
#ifdef PLATFORM_A
extern uint32_t num_lives;
#else
extern uint16_t num_lives;
#endif
```

Когда `screen.c` подключает его, тип импортируемой сущности `num_lives` зависит от определения макроса `PLATFORM_A`, который может или не может быть определен в контексте единицы трансляции `screen.c`. Кроме того, этот контекст необязательно будет совпадать с контекстом в другой единице трансляции. Данное обстоятельство вынуждает компилятор загружать дополнительный код из заголовочных файлов всякий раз, когда другая единица трансляции подключает его.

Чтобы укротить проблему импортирования в C/C++ и изменить порядок оформления интерфейсов библиотек, модули предлагают новый способ описания интерфейсов, о чем сейчас ведутся бурные дискуссии. Кроме того, проект Clang уже включает поддержку модулей.

Модули

Вместо подключения заголовочных файлов, единица трансляции может импортировать модуль, который определяет ясный и однозначный интерфейс к некоторой библиотеке. Директива `import` может загружать сущности, экспортируемые указанной библиотекой без внедрения в нее дополнительного кода на C или C++.

Однако, синтаксис импортирования пока не определен и активно обсуждается в комитете по стандартизации C++. В настоящее время Clang позволяет передавать дополнительный флаг `-fmodules`, заставляющий компилятор интерпретировать директивы `include` как директивы импорта модулей, если подключаемые заголовочные файлы принадлежат библиотеке, поддерживающей модули.

Выполняя парсинг заголовочных файлов, принадлежащих модулям, Clang порождает новый экземпляр самого себя, специально для компиляции этих заголовочных файлов, и кэширует результат в двоичной форме, чтобы ускорить компиляцию последующих единиц трансляции, зависящих от тех же самых заголовочных файлов. Соответственно, заголовочные файлы, являющиеся частью модуля, не должны зависеть от внешних макросов или состояния препроцессора.

Использование модулей

Чтобы отобразить множество заголовочных файлов на конкретный модуль, можно создать отдельный файл с именем `module.modulemap`, куда поместить всю необходимую информацию. Этот файл должен находиться в том же каталоге, что и подключаемые файлы с определением API библиотеки. Если такой файл присутствует и Clang вызывается с ключом `-fmodules`, компиляция будет выполняться с использованием модулей. Давайте дополним предыдущий пример поддержкой модулей. Допустим, что API определяется двумя заголовочными файлами, `gamelogic.h` и `screenlogic.h`. Основной файл программы, `game.c`, импортирует сущности из обоих файлов. Исходный код API содержит следующие определения:

- файл `gamelogic.h`:

```
extern int num_lives;
```
- файл `screenlogic.h`:

```
extern int num_lines;
```
- файл `gamelogic.c`:

```
int num_lives = 3;
```
- файл `screenlogic.c`:

```
int num_lines = 24;
```

Кроме того, когда пользователь подключает заголовочный файл `gamelogic.h`, он должен также подключить `screenlogic.h`, чтобы выводить игровые данные на экран. Поэтому определим структуру

наших модулей так, чтобы она отражала эту зависимость. Ниже приводится содержимое получившегося файла `module.modulemap` для нашего проекта:

```
module MyGameLib {
    explicit module ScreenLogic {
        header "screenlogic.h"
    }
    explicit module GameLogic {
        header "gamelogic.h"
        export ScreenLogic
    }
}
```

Ключевое слово `module` сопровождается именем, которое будет идентифицировать модуль. В данном случае мы дали модулю имя `MyGameLib`. Каждый модуль может иметь список вложенных субмодулей. Ключевое слово `explicit` сообщает компилятору Clang, что данный субмодуль импортируется, только если его заголовочные файлы подключаются явно. Далее используется ключевое слово `header` с именем заголовочного файла, образующего данный субмодуль. Для представления одного субмодуля можно указать несколько заголовочных файлов, но в данном случае для определения субмодулей мы использовали по одному заголовочному файлу.

Решив использовать концепцию модулей, мы можем воспользоваться дополнительными их преимуществами и упростить директивы `include`. Обратите внимание, что субмодулю `GameLogic` присутствует ключевое слово `export` с именем субмодуля `ScreenLogic`. Мы уже говорили, что при импортировании субмодуля `GameLogic` пользователь должен также получить доступ к символам в субмодуле `ScreenLogic`.

Для демонстрации напомним файл `game.c`, использующий данный API:

```
// Файл: game.c
#include "gamelogic.h"
#include <stdio.h>

int main() {
    printf("lives= %d\nlines=%d\n", num_lives, num_lines);
    return 0;
}
```

Обратите внимание, что программа использует символ `num_lives`, объявленный в `gamelogic.h`, и символ `num_lines`, объявленный в `screenlogic.h`, причем заголовочный файл `screenlogic.h` не под-

ключается к программе явно. Однако, когда компилятор `clang` вызывается с флагом `-fmodules`, он просмотрит этот файл, преобразует первую директиву `include` в директиву импортирования субмодуля `GameLogic`, который в свою очередь требует сделать доступными символы из модуля `ScreenLogic`. То есть, следующая команда вполне корректно скомпилирует этот проект:

```
$ clang -fmodules game.c gamelogic.c screenlogic.c -o game
```

С другой стороны, попытка вызвать `Clang` без поддержки модулей приведет к появлению ошибки, сообщающей об отсутствии определения символа:

```
$ clang game.c gamelogic.c screenlogic.c -o game

screen.c:4:50: error: use of undeclared identifier 'num_lines';
did you mean 'num_lives'?
    printf("lives= %d\nlines=%d\n", num_lives, num_lines);
                                                ^~~~~~
                                                num_lives
```

Имейте в виду, что если вам требуется обеспечить максимальную переносимость своих проектов, тогда, чтобы избежать подобных ситуаций, модули должны быть организованы так, чтобы проект корректно компилировался как с их применением, так и без них. Модули лучше всего подходят в ситуациях, когда их применение упрощает использование API библиотеки и ускоряет компиляцию единиц трансляции, использующих множество общих заголовочных файлов.

Modularize

Отличным примером может послужить попытка адаптации существующего большого проекта под использование модулей. Главное помнить, что при использовании модулей заголовочные файлы, принадлежащие субмодулям, компилируются независимо. Многие проекты используют макросы, которые определяются перед подключением заголовочных файлов. Такие проекты невозможно перевести на использование модулей.

Основной целью инструмента `modularize` является оказание помощи в решении задачи внедрения модулей. Он анализирует множество заголовочных файлов и сообщает, если обнаруживает повторяющиеся определения переменных, макросов или определения макросов, которые могут приводить к разным результатам, в зависимости от состояния препроцессора. Он поможет вам выявить типичные проблемы, препятствующие созданию модулей на множестве

заголовочных файлов. Он также определяет ситуации, когда проект использует директивы `include` внутри пространств имен, из-за чего компилятор вынужден интерпретировать подключаемые файлы в разных контекстах, что так же мешает применению модулей. Для успешного перехода на использование модулей заголовочные файлы не должны зависеть от контекста подключения.

Практика использования инструмента `modularize`

Чтобы воспользоваться инструментом `modularize`, необходимо подготовить список заголовочных файлов, которые он должен про-версить. Продолжим наш пример с проектом игровой программы и создадим новый текстовый файл с именем `list.txt`:

```
gamelogic.h  
screenlogic.h
```

Теперь достаточно просто запустить `modularize` и передать ему имя файла со списком:

```
$ modularize list.txt
```

Если изменить один из заголовочных файлов, включив в него определение символа, присутствующего в другом заголовочном файле, `modularize` сообщит, что проект опирается на небезопасное для модулей поведение, и что следует исправить проблему перед повторной попыткой создать файл `module.modulemap` для проекта. Исправляя заголовочные файлы, имейте в виду, что каждый заголовочный файл должен быть максимально независимым и не должен изменять определения символов, в зависимости от других определений в подключаемых им других заголовочных файлах. Если вы полагаетесь на подобное, контекстно-зависимое поведение, такие заголовочные файлы следует разделить на два или более файлов, каждый из которых компилятор будет видеть при использовании определенного набора макросов.

Module Map Checker

Инструмент `Module Map Checker` исследует файл `module.modulemap`, проверяя – охватывает ли он все заголовочные файлы в каталоге. Для примера из предыдущего раздела этот инструмент можно вызвать так:

```
$ module-map-checker module.modulemap
```

Центральное место в нашем обсуждении директив `include` и моду-

лей занимал препроцессор. В следующем разделе мы познакомим вас с инструментом, оказывающим помощь в трассировке деятельности этого специфического компонента анализатора исходных текстов.

PPTrace

Взгляните на следующую цитату из документации к Clang, относящуюся к `clang::preprocessor` (http://clang.llvm.org/doxygen/classclang_1_1Preprocessor.html):

Таңуует в тесном контакте с лексическим анализатором, чтобы обеспечить эффективную предварительную обработку лексем.

Как уже отмечалось в главе 4, «Анализатор исходного кода», класс `lexer` в Clang выполняет первый проход в анализе исходных файлов. Он объединяет фрагменты текста в категории для последующей интерпретации парсером. Класс `lexer` не имеет информации о семантике, за которую отвечает парсер, а также о подключаемых заголовочных файлах и используемых макросах, за которые отвечает препроцессор.

Инструмент `pp-trace` выводит трассировочную информацию о работе препроцессора. Достигается это за счет реализации функций обратного вызова в интерфейсе `clang::PPCallbacks`. Он начинает с регистрации самого себя в роли «наблюдателя» за препроцессором и затем запускает анализ файлов. Информация о всех действиях препроцессора, таких как интерпретация директивы `#if`, импортирование модуля, подключение заголовочного файла и многих других, выводится на экран.

Взгляните на следующий пример программы «Hello world» на языке C:

```
#if 0
#include <stdio.h>
#endif

#ifdef CAPITALIZE
#define WORLD "WORLD"
#else
#define WORLD "world"
#endif

extern int write(int, const char*, unsigned long);

int main() {
```

```

write(1, "Hello, ", 7);
write(1, WORLD, 5);
write(1, "!\n", 2);
return 0;
}

```

В первых строках используется директива препроцессора `#if`, которая всегда вычисляется как «ложь», вынуждая компилятор игнорировать любой код до следующей директивы `#endif`. Далее используется директива `#ifdef`, которая проверяет – определен ли макрос `CAPITALIZE`. В зависимости результата проверки, макрос `WORLD` определяется как строка со словом «world», состоящим из символов верхнего или нижнего регистра. В конце выполняется последовательность обращений к системному вызову `write` для вывода сообщения на экран.

Запускается `pp-trace` точно так же, как другие подобные инструменты Clang, предназначенные для обработки исходных текстов:

```
$ pp-trace hello.c
```

В результате возникает серия событий препроцессора, касающихся определения макросов, еще до того, как начнется фактическая обработка исходного файла. Ниже приводятся последние события, возникшие при обработке нашего файла:

```

- Callback: If
  Loc: "hello.c:1:2"
  ConditionRange: ["hello.c:1:4", "hello.c:2:1"]
  ConditionValue: CVK_False
- Callback: Endif
  Loc: "hello.c:3:2"
  IfLoc: "hello.c:1:2"
- Callback: SourceRangeSkipped
  Range: ["hello.c:1:2", "hello.c:3:2"]
- Callback: Ifdef
  Loc: "hello.c:5:2"
  MacroNameTok: CAPITALIZE
  MacroDirective: (null)
- Callback: Else
  Loc: "hello.c:7:2"
  IfLoc: "hello.c:5:2"
- Callback: SourceRangeSkipped
  Range: ["hello.c:5:2", "hello.c:7:2"]
- Callback: MacroDefined
  MacroNameTok: WORLD
  MacroDirective: MD_Define
- Callback: Endif
  Loc: "hello.c:9:2"

```

```
IfLoc: "hello.c:5:2"  
- Callback: MacroExpands  
  MacroNameTok: WORLD  
  MacroDirective: MD_Define  
  Range: ["hello.c:13:14", "hello.c:13:14"]  
  Args: (null)  
- Callback: EndOfFile
```

Первое событие относится к первой директиве препроцессора `#if`. В этой области генерируется три события: `If`, `Endif` и `SourceRangeSkipped`. Обратите внимание, что директива `#include` внутри не была обработана. Аналогично, далее следуют события, имеющие отношение к определению макроса `WORLD`: `IfDef`, `Else`, `MacroDefined` и `EndIf`. Наконец, `pp-trace` сообщает об использовании макроса `WORLD` с помощью события `MacroExpands`, после чего он достигает конца файла и генерирует событие `EndOfFile`.

Следующий этап после предварительной обработки – это лексический и синтаксический анализ. В следующем разделе мы познакомимся с инструментом, который позволяет получить результаты парсинга: узлы абстрактного синтаксического дерева AST.

Clang Query

Инструмент `Clang Query` впервые появился в версии LLVM 3.5. Он позволяет читать исходные файлы и интерактивно запрашивать узлы дерева AST. Это отличный инструмент для исследований и получения информации о том, как анализатор исходных текстов представляет каждый фрагмент кода. Однако, главная его цель не только в том, чтобы дать возможность исследовать дерево AST программы, но и отыскивать его сегменты по заданным условиям.

При создании инструментов рефакторинга вас наверняка заинтересует библиотека с коллекцией предикатов, которые определяют соответствие сегментов дерева AST определенным условиям. `Clang Query` специально создавался в помощь разработчикам – он позволяет выявлять узлы AST, соответствующие заданным условиям. Список доступных предикатов можно найти в заголовочном файле `ASTMatchers.h`. Заметим, что по именам классов в этой библиотеке, записанным в «верблюжьей» нотации, легко догадаться, с какими узлами дерева AST они совпадают. Например, `functionDecl` соответствует всем узлам `FunctionDecl`, представляющим объявления функций. Выяснив, какие узлы возвращают те или иные предикаты, вы сможете использовать их в своем инструменте рефакторинга для

автоматизации преобразований с той или иной целью. Порядок использования библиотеки предикатов AST мы опишем далее в этой главе.

Для примера попробуем применить инструмент `clang-query` к программе «Hello world» из раздела с описанием PPTrace. Clang Query предполагает наличие базы данных команд компиляции. Если вы собираетесь исследовать файл в отсутствие такой базы данных, укажите параметры компилятора после двойного дефиса или не указывайте их вообще, если для компиляции не требуется никаких дополнительных флагов, как показано ниже:

```
$ clang-query hello.c --
```

После запуска, `clang-query` выведет интерактивное приглашение к вводу, ожидая ваших команд. Теперь можно вести команду `match` и имя предиката. Например, следующая команда требует от `clang-query` вывести все узлы типа `CallExpr`:

```
clang-query> match callExpr()
```

```
Match #1:
```

```
hello.c:12:5: note: "root" node binds here
    write(1, "Hello, ", 7);
    ^~~~~~
```

```
...
```

Как видите, инструмент вывел точное местоположение в программе лексемы, соответствующей узлу `CallExpr` в дереве AST. Ниже приводится список команд Clang Query:

- `help`: выводит список доступных команд;
- `match <имя предиката>` или `m <имя предиката>`: выполняет обход дерева AST и выполняет поиск узлов, соответствующих указанному предикату;
- `set output <(diag | print | dump)>`: определяет, как будет выводиться информация о найденных узлах. Параметр `diag` (действует по умолчанию) обеспечивает вывод диагностических сообщений для найденных узлов. Параметр `print` обеспечивает простой вывод соответствующего фрагмента исходного кода. А параметр `dump` обеспечивает вызов метода `dump()`, который дополнительно выводит все дочерние узлы.

Отличный способ узнать, как структурировано дерево AST программы состоит в том, чтобы установить параметр `dump` и выполнить поиск узлов верхнего уровня. Попробуйте:

```
clang-query> set output dump  
clang-query> match functionDecl()
```

В результате вы получите все экземпляры классов, составляющие инструкции и выражения во всех функциях, которые определены в исследуемом исходном файле. Но, имейте в виду, что то же самое дерево AST проще получить с помощью инструмента Clang Check, с которым мы познакомимся в следующем разделе. Clang Query больше подходит для выполнения запросов к дереву AST и исследования их результатов. Позднее, если вы займетесь созданием собственного инструмента рефакторинга, у вас будет возможность оценить по достоинству инструмент Clang Query, особенно когда попытаете конструировать более сложные запросы, чем было показано выше.

Clang Check

Инструмент Clang Check очень прост; его реализация насчитывает всего несколько сот строк кода, и вы без труда сможете разобраться в ней самостоятельно. Однако, благодаря использованию фреймворка LibTooling, он обладает всей полнотой возможностей парсера Clang.

Clang Check позволяет выполнять парсинг исходного кода на C/C++ и выводить дерево AST или выполнять простые проверки. Он также может применять «исправления», предлагаемые Clang, используя для этого инфраструктуру, построенную на основе Clang Modernizer.

Например, представьте, что вам требуется получить дерево AST для файла `program.c`. Для этого достаточно выполнить следующую команду:

```
$ clang-check program.c -ast-dump --
```

Обратите внимание, что Clang Check следует соглашениям, принятым в LibTooling, в отношении базы данных команд компиляции: вы должны указать файл с базой данных или добавить в командную строку параметры компиляции после двойного дефиса (`--`).

Так как Clang Check – небольшой инструмент, он может послужить отличным примером для изучения, если вы решите заняться созданием собственных инструментов на основе LibTooling. В следующем разделе мы расскажем вам о еще одном небольшом инструменте, чтобы вы могли увидеть, насколько маленькими могут быть инструменты для рефакторинга кода.

Удаление вызовов `c_str()`

Инструмент `remove-cstr-calls` – это простой пример инструмента преобразования исходных текстов, то есть, инструмента рефакторинга. Он ищет избыточные вызовы метода `c_str()` объектов `std::string` и удаляет их. Такие вызовы считаются избыточными, во-первых, когда новый объект `string` создается с вызовом метода `c_str()` другого объекта `string`, например: `std::string(myString.c_str())`. Эту операцию можно упростить за счет непосредственного использования конструктора копии: `std::string(myString)`. Во-вторых, когда на основе объектов `string` создаются новые экземпляры классов LLVM `StringRef` и `Twine`. В этих случаях предпочтительнее использовать сам объект `string`, а не результат вызова `c_str()`, то есть: `StringRef(myString)` вместо `StringRef(myString.c_str())`.

Реализация инструмента на C++ целиком уместилась в единственный файл, что делает его еще одним замечательным примером для самостоятельного изучения приемов использования фреймворка LibTooling, знание которых пригодится при создании собственного инструмента рефакторинга, о чем и пойдет речь в следующем разделе.

Создание собственного инструмента

Проект Clang предоставляет три интерфейса для доступа к функциональным возможностям Clang и средствам синтаксического анализа, включая синтаксический и семантический анализ. Первый из них – библиотека `libclang`, которая является основным интерфейсом к механизмам Clang и позволяет использовать высокоуровневые абстракции для доступа к ним. Этот интерфейс поддерживает обратную совместимость, обеспечивая работоспособность существующего программного обеспечения после выхода новой версии `libclang`. Библиотеку `libclang` можно также использовать из программ на других языках, посредством соответствующей промежуточной библиотеки, такой как `Clang Python Bindings`. `Apple Xcode`, к примеру, взаимодействует с Clang через интерфейс `libclang`.

Второй интерфейс – `Clang Plugins` – позволяет добавлять свои проходы непосредственно в процедуру компиляции, в отличие от других инструментов анализа, таких как статический анализатор `Clang Static Analyzer`. Такая возможность может пригодиться тем, кому требу-



ется организовать выполнение определенных операций при каждой компиляции единицы трансляции. При этом придется уделить особое внимание скорости выполнения своего анализа, чтобы не сильно тормозить процедуру компиляции. С другой стороны, можно предусмотреть запуск своего анализа в процессе компиляции, только если пользователь указал специальный флаг.

И третий интерфейс, исследованием которого мы займемся в этом разделе – фреймворк `LibTooling`. Эта замечательная библиотека позволяет создавать самостоятельные инструменты, подобные тем, что были представлены выше в этой главе, предназначенные для выполнения рефакторинга кода или проверки синтаксиса. В отличие от `libclang`, фреймворк `LibTooling` менее надежен с точки зрения поддержки обратной совместимости, но зато дает полный доступ к структуре дерева `AST`.

Определение задачи – создание инструмента рефакторинга кода на C++

В оставшейся части главы мы займемся разработкой своего инструмента на основе `LibTooling`. Допустим, что мы решили заняться созданием своей интегрированной среды разработки программ на `C++`, которая называется `IzzyC++`. Наша цель – привлечь пользователей, ищущих возможность автоматического рефакторинга кода. Для создания простого и удобного инструмента рефакторинга мы будем использовать фреймворк `LibTooling`; он будет принимать в виде параметров исходное и целевое имя метода, и замещать исходное имя целевым. Задача заключается в том, чтобы найти определение исходного метода, заменить его имя целевым и изменить все обращения к исходному имени обращением у целевому имени.

Определение структуры каталогов для исходного кода

Первым делом необходимо определиться с размещением исходного кода инструмента. В дереве с исходными текстами `LLVM`, внутри подкаталога `tools/clang/tools/extra`, создайте новый каталог с именем `izzyrefactor`, где будут храниться исходные файлы проекта. Затем отредактируйте `Makefile` в каталоге `extra` и включите в него наш проект. Для этого достаточно добавить имя каталога проекта в переменную `DIRS` и имя `izzyrefactor` в список с другими

проектами инструментов Clang. Можно также отредактировать файл `CMakeLists.txt`, если предполагается использовать CMake, и включить в него новую строку:

```
add_subdirectory(izyrefactor)
```

Перейдите в каталог `izyrefactor` и создайте новый файл `Makefile`, чтобы подсказать системе сборки LLVM, что это отдельный инструмент, который должен компилироваться в собственный выполняемый файл:

```
CLANG_LEVEL := ../../..
TOOLNAME = izyrefactor
TOOL_NO_EXPORTS = 1
include $(CLANG_LEVEL)/../../Makefile.config
LINK_COMPONENTS := $(TARGETS_TO_BUILD) asmparser bitreader support\
mc option
USEDLIBS = clangTooling.a clangFrontend.a clangSerialization.a \
clangDriver.a clangRewriteFrontend.a clangRewriteCore.a \
clangParse.a clangSema.a clangAnalysis.a clangAST.a \
clangASTMatchers.a clangEdit.a clangLex.a clangBasic.a
include $(CLANG_LEVEL)/Makefile
```

Этот файл играет важную роль, определяя все библиотеки, с которыми необходимо скомпоновать наш код. Дополнительно можно добавить строку `NO_INSTALL = 1` сразу после строки `TOOL_NO_EXPORTS = 1`, если не требуется устанавливать новый инструмент вместе с другими инструментами LLVM при выполнении команды `make install`.

Мы определили переменную `TOOL_NO_EXPORTS = 1`, потому что данный инструмент не будет использовать никаких расширений и потому ему не требуется экспортировать какие-либо символы. Это позволяет уменьшить в размерах динамическую таблицу символов в конечном выполняемом файле и, соответственно, время, необходимое на динамическую компоновку во время запуска программы. Обратите внимание, что в конце подключается основной файл `Makefile` проекта Clang, где определяются все правила, необходимые для компиляции проекта.

Если предполагается использовать систему сборки CMake вместо сценария `configure`, создайте новый файл `CMakeLists.txt` со следующим содержанием:

```
add_clang_executable(izyrefactor
  IzzyRefactor.cpp
)
target_link_libraries(izyrefactor
  clangEdit clangTooling clangBasic clangAST clangASTMatchers)
```

Если же у вас нет желания собирать инструмент внутри дерева с исходными текстами Clang, можно организовать его сборку отдельно. Просто используйте тот же Makefile, представленный в конце главы 4, «Анализатор исходного кода», для сборки драйвер, внося в него небольшие изменения. Обратите внимание, какие библиотеки использовались в предыдущем Makefile, в переменной USEDLIBS, и какие библиотеки используются в Makefile из главы 4, «Анализатор исходного кода», в переменной CLANGLIBS. Списки библиотек практически идентичны, за исключением clangTooling, включенной в список USEDLIBS. Соответственно, добавьте строку `-lclangTooling\` после строки `-lclang\` в Makefile из главы 4, «Анализатор исходного кода», и дело сделано.

Шаблонный код инструмента

Вся реализация инструмента будет находиться в единственном файле `IzzyRefactor.cpp`. Создайте этот файл и добавьте в него следующий шаблонный код:

```
int main(int argc, char **argv) {
    cl::ParseCommandLineOptions(argc, argv);
    string ErrorMessage;
    OwningPtr<CompilationDatabase> Compilations (
        CompilationDatabase::loadFromDirectory(
            BuildPath, ErrorMessage));
    if (!Compilations)
        report_fatal_error(ErrorMessage);
    //...
}
```

Функция `main` начинается с вызова функции `ParseCommandLineOptions` из пространства имен `llvm::cl` (утилиты обработки параметров командной строки). Эта функция выполнит всю грязную работу по анализу отдельных флагов в `argv`.

Совет. Для инструментов на основе `LibTooling` принято использовать объект `CommonOptionsParser`, чтобы упростить парсинг типичных параметров командной строки, используемых всеми инструментами рефакторинга (см. пример кода по адресу: http://clang.llvm.org/doxygen/classclang_1_1_tooling_1_1_CommonOptionsParser.html). В данном примере мы использовали низкоуровневую функцию `ParseCommandLineOptions()`, чтобы показать, какие именно параметры мы будем анализировать, и научить вас пользоваться этим приемом в других инструментах, не использующих фреймворк `LibTooling`. Однако вы можете использовать объект `CommonOptionsParser`, чтобы упростить себе работу (и в качестве упражнения по созданию того же инструмента другим способом).

Вы можете заглянуть в другие инструменты LLVM и убедиться, что все они используют утилиты из пространства имен `cl` (http://llvm.org/docs/Doxygen/html/namespacellvm_1_1cl.html). С их помощью действительно очень просто определить, какие параметры должен распознавать инструмент. Для этого требуется объявить новые глобальные переменные с шаблонными типами `opt` и `list`:

```
cl::opt<string> BuildPath(
    cl::Positional,
    cl::desc("<build-path>"));
cl::list<string> SourcePaths(
    cl::Positional,
    cl::desc("<source0> [... <sourceN>]"),
    cl::OneOrMore);
cl::opt<string> OriginalMethodName("method",
    cl::desc("Method name to replace"),
    cl::ValueRequired);
cl::opt<string> ClassName("class",
    cl::desc("Name of the class that has this method"),
    cl::ValueRequired);
cl::opt<string> NewMethodName("newname",
    cl::desc("New method name"),
    cl::ValueRequired);
```

Вставьте эти объявления перед функцией `main`. Тип `opt` должен специализироваться соответствующими типами параметров. Например, если инструмент должен принимать параметр как целое число, следует объявить новую глобальную переменную типа `cl::opt<int>`.

Чтобы прочитать значения параметров, сначала необходимо вызвать `ParseCommandLineOptions`. А затем достаточно просто ссылаться на имя глобальной переменной, представляющей этот параметр. Например, переменная `NewMethodName` будет возвращать строку, переданную пользователем в соответствующем параметре, которую легко можно вывести инструкцией `std::out << NewMethodName`.

Такое возможно благодаря тому, что шаблон `opt_storage<>`, суперкласс для `opt<>`, определяет класс, который наследует управляемый им тип данных (в данном случае `string`). в соответствии с правилами наследования, переменная типа `opt<string>` также является переменной типа `string` и может использоваться соответственно этому типу. Если шаблонный класс `opt<>` не может наследовать обертываемый им тип данных (например, в C++ нет класса `int`), он определит оператор приведения типа, например, оператор `int()` для типа `int`. Это дает тот же эффект; при обращении к переменной типа

`cl::opt<int>`, ее значение автоматически приводится к целочисленному типу и возвращается целое число, переданное пользователем в параметре командной строки.

Имеется также возможность описать другие характеристики параметров. В нашем примере с помощью `cl::Positional` определяется позиционный аргумент. То есть пользователь не сможет передать его значение по имени. Семантика позиционных аргументов определяется их местоположениями в командной строке. Конструктору `opt` передается также объект `desc` с описанием, которое будет выведено на экран, если пользователь вызовет инструмент с флагом `-help`.

У нас также есть параметр типа `cl::list`, который отличается от `opt` возможностью передачи нескольких значений, в данном случае – список исходных файлов для обработки. Эти возможности требуют подключения заголовочного файла:

```
#include "llvm/Support/CommandLine.h"
```

Совет. Согласно стандартам LLVM, локальные заголовочные файлы должны подключаться первыми, и только затем должны подключаться заголовочные файлы Clang и LLVM. Когда два файла относятся к одной категории, они должны подключаться в алфавитном порядке. Было бы интересно написать новый автономный инструмент, который бы гарантировал соблюдение этого правила.

Последние три глобальные переменные позволяют прочесть параметры командной строки, необходимые инструменту для работы. Первая соответствует параметру с именем `-method`. Первая строка в определении объявляет имя параметра без дефиса, а вторая, `cl::RequiredValues`, сообщает парсеру, что этот параметр является обязательным. Через этот параметр передается имя метода, которое наш инструмент должен найти и заменить именем из параметра `-newname`. Параметр `-class` определяет имя класса, которому принадлежит метод.

Следующий фрагмент в шаблонном коде управляет новым объектом `CompilationDatabase`. Для его работы необходимо сначала подключить заголовочные файлы с определением класса `OwningPtr`, который реализует интеллектуальные указатели, широко используемые в библиотеках LLVM, то есть указатели, автоматически освобождающие память по достижении конца своей области видимости.

```
#include "llvm/ADT/OwningPtr.h"
```


Примечание. Начиная с версии Clang/LLVM 3.5, вместо шаблона `OwningPtr<>` рекомендуется использовать стандартный шаблон `std::unique_ptr<>`.

Затем нужно подключить заголовочный файл с определением класса `CompilationDatabase`, который является официальной частью фреймворка `LibTooling`:

```
#include "clang/Tooling/CompilationDatabase.h"
```

Этот класс отвечает за управление базой данных команд компиляции, об устройстве которой рассказывалось в начале главы. Это список команд компиляции для обработки каждого файла с исходным кодом, который пользователь хотел бы исследовать с помощью вашего инструмента. Для инициализации этого объекта мы использовали фабричный метод `loadFromDirectory`, который загружает базу данных из файла в указанном каталоге сборки. Именно с этой целью мы объявили параметр `build-path`; пользователь должен указать, где находятся исследуемые файлы и файл базы данных команд компиляции.

Обратите внимание на два аргумента, что передаются фабричному методу: `BuildPath`, наш объект `cl::opt`, представляющий параметр командной строки, и объявленная в функции `main` строковая переменная `ErrorMessage`. Если при попытке загрузить файл базы данных возникнет ошибка, в эту переменную будет записана строка с текстом сообщения об ошибке, которая тут же выводится, если обнаружится, что фабричный метод не вернул объект `CompilationDatabase`. Функция `llvm::report_fatal_error()` вызывает все установленные процедуры обработки ошибок и завершает инструмент с кодом ошибки 1. Для поддержки обработки ошибок необходимо подключить следующий заголовочный файл:

```
#include "llvm/Support/ErrorHandler.h"
```

Поскольку в нашем примере мы сократили полные имена многих классов, необходимо также добавить несколько объявлений `using`, но вы можете использовать полные квалифицированные имена, если вам так пожелается:

```
using namespace clang;
using namespace std;
using namespace llvm;
using clang::tooling::RefactoringTool;
using clang::tooling::Replacement;
```

```
using clang::tooling::CompilationDatabase;  
using clang::tooling::newFrontendActionFactory;
```

Использование предикатов AST

Предикаты AST были кратко описаны в разделе «Clang Query», выше, но здесь мы рассмотрим их подробнее, потому что они играют очень важную роль в инструментах рефакторинга на основе Clang.

Библиотека предикатов AST позволяет пользователям легко находить поддеревья в дереве AST, соответствующие заданным условиям, например, все узлы AST, представляющие вызов функции с именем `calloc` и двумя аргументами. Поиск определенных узлов AST и их изменение являются фундаментальной задачей, используемой многими инструментами рефакторинга, и применение этой библиотеки существенно упрощает разработку таких инструментов.

Чтобы определиться с выбором предикатов, которые помогли бы решить поставленную задачу, мы использовали документацию с описанием инструмента Clang Query и предикатов AST, доступную по адресу: <http://clang.llvm.org/docs/LibASTMatchersReference.html>.

Начнем с создания тестового примера для отладки нашего инструмента (`wildlifesim.cpp`). Это сложный, одномерный имитатор животных, в котором животные могут перемещаться в любом направлении вдоль прямой линии:

```
class Animal {  
    int position;  
public:  
    Animal(int pos) : position(pos) {}  
    // Возвращает новое местоположение  
    int walk(int quantity) {  
        return position += quantity;  
    }  
};  
  
class Cat : public Animal {  
public:  
    Cat(int pos) : Animal(pos) {}  
    void meow() {}  
    void destroySofa() {}  
    bool wildMood() {return true;}  
};  
  
int main() {  
    Cat c(50);  
    c.meow();  
    if (c.wildMood())
```

```

    c.destroySofa();
    c.walk(2);
    return 0;
}

```

Представим, что с помощью нашего инструмента мы хотели бы переименовать, например, метод `walk` в `run`. Начнем с того, что запустим `Clang Query` и посмотрим, как выглядит дерево AST для этого примера. Воспользуемся предикатом `recordDecl` и выведем содержимое всех узлов `RecordDecl`, которые представляют структуры `C` и классы `C++`:

```

$ clang-query wildanimal-sim.cpp --
clang-query> set output dump
clang-query> match recordDecl()

(...)
|---CXXMethodDecl 0x(...) <line:6:3, line 8:3> line 6:7 walk 'int (int)'
(...)

```

Внутри объекта `RecordDecl`, представляющего класс `Animal` метод `walk` представлен, как узел `CXXMethodDecl`. Заглянув в документацию, мы увидели, что узлам этого типа соответствует предикат `methodDecl`.

Комбинирование предикатов

Мощь предикатов AST во многом объясняется возможностью комбинировать их. Например, если требуется найти только узлы `MethodDecl`, представляющие метод с именем `walk`, можно сначала отобразить все именованные объявления с именем `walk` и затем оставить только те из них, которые являются объявлениями методов. Предикат `hasName("input")` возвращает все именованные объявления с именем "input". Проверить, как действует комбинация предикатов `methodDecl` и `hasName`, можно с помощью `Clang Query`:

```

clang-query> match methodDecl(hasName("walk"))

```

В результате, вместо списка из всех восьми объявлений методов, присутствующих в тестовом примере, возвращается только одно — объявление метода `walk`. Отлично!

Но, как бы то ни было, недостаточно изменить определение метода `walk` только в классе `Animal`, потому что дочерние классы могут переопределять его. Нам бы не хотелось, чтобы инструмент рефакторинга заменил имя метода в родительском классе, оставив неизменными переопределенные реализации в дочерних классах.

Итак, нужно найти все классы с именем `Animal` или наследующие его, которые определяют метод `walk`. Для этого можно воспользоваться предикатом `isSameOrDerivedFrom()`, который принимает `NamedDecl` в качестве параметра. Этот параметр будет получен комбинацией предикатов, отбирающих узлы `NamedDecl` с указанным именем. То есть, требуемый запрос будет выглядеть так:

```
clang-query> match recordDecl(isSameOrDerivedFrom  
(hasName("Animal")))
```

Нам также потребуется отобрать только дочерние классы, определяющие метод `walk`. Решить эту задачу можно с помощью предиката `hasMethod()`, возвращающего объявления классов, содержащих указанный метод. Скомбинируем его с предыдущим запросом:

```
clang-query> match recordDecl(hasMethod(methodDecl(  
hasName("walk"))))
```

Чтобы объединить два предиката с семантикой оператора `and` (должны выполняться все условия, определяемые предикатами), действуем предикат `allOf()`. Он требует соответствия со всеми предикатами, которые передаются ему в качестве аргументов. Теперь все готово к созданию запроса в окончательном виде, который отыскивает все объявления, подлежащие замене:

```
clang-query> match recordDecl(allOf(hasMethod(methodDecl  
(hasName("walk"))),  
isSameOrDerivedFrom(hasName("Animal"))))
```

С этим запросом мы теперь сможем определить точное местоположение всех объявлений методов `walk` в классах с именем `Animal` или наследующих его.

Это позволит нам изменить все объявленные имена, но нам еще нужно изменить имена в вызовах метода. Для этого сначала отберем все узлы `SXXMemberCallExpr` с помощью предиката `memberCallExpr`. Попробуйте:

```
clang-query> match memberCallExpr()
```

В результате Clang Query вернет четыре совпадения, потому что наш код делает точно четыре вызова методов: `meow`, `wildMood`, `destroySofa` и `walk`. Нас интересует только вызов последнего из них. Мы уже знаем, как выбирать именованные объявления с помощью предиката `hasName()`, но как отобразить именованные объявления в выражения вызовов членов? Для этого достаточно воспользоваться предикатом `member()`, который отбирает только именованные

объявления, связанные с именем метода, и затем вызвать предикат `callee()`, чтобы связать результаты с выражением вызова. Полный запрос имеет вид:

```
clang-query> match memberCallExpr (callee (memberExpr (member
(hasName ("walk")))))
```

Однако этот запрос слепо отберет все вызовы метода `walk()`, а нам нужны только вызовы метода `walk`, принадлежащего классу `Animal` или его дочерним классам. Предикат `memberCallExpr()` может принимать второй предикат в виде аргумента. Поэтому мы передадим ему предикат `thisPointerType()` для выбора только вызовов метода, принадлежащих определенному классу. Используя этот принцип мы сконструировали более полный запрос:

```
clang-query> match memberCallExpr (callee (memberExpr (member
(hasName ("walk")))),
thisPointerType (recordDecl (isSameOrDerivedFrom (hasName
("Animal")))))
```

Включение предикатов AST в программный код

Теперь, когда мы определили, какие предикаты использовать для выбора требуемых узлов дерева AST, их нужно включить в программный код инструмента. Чтобы воспользоваться предикатами, следует добавить новые директивы `include`:

```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
```

Также нужно добавить еще одну директиву `using`, чтобы упростить ссылки на классы (поместите ее после других директив `using`):

```
using namespace clang::ast_matchers;
```

Второй заголовочный файл необходим, чтобы получить возможность использовать фактический механизм поиска, с которым мы познакомимся ниже. Итак, продолжим писать функцию `main` с того места, где мы оставили:

```
RefactoringTool Tool(*Compilations, SourcePaths);
ast_matchers::MatchFinder Finder;
ChangeMemberDecl DeclCallback(&Tool.getReplacements());
ChangeMemberCall CallCallback(&Tool.getReplacements());
Finder.addMatcher(
    recordDecl(
        allOf(hasMethod(id("methodDecl",
                        methodDecl(hasName(OriginalMethodName)))),
              isSameOrDerivedFrom(hasName(ClassName)))),
```

```
&DeclCallback);
Finder.addMatcher(
    memberCallExpr(
        callee(id("member",
                memberExpr(member(hasName(OriginalMethodName))))),
        thisPointerType(recordDecl(
            isSameOrDerivedFrom(hasName(Classname))))),
    &CallCallback);
return Tool.runAndSave(newFrontendActionFactory(&Finder)););
```

Примечание. При использовании версии Clang 3.5 необходимо заменить последнюю строку в предыдущем фрагменте на:

```
return Tool.runAndSave(newFrontendActionFactory
                        (&Finder).get());
```

На этом реализацию функции `main` можно считать завершенной. Далее мы покажем реализации обработчиков, или функций обратного вызова.

Первая строка в этом фрагменте создает новый объект `RefactoringTool`. Это второй класс из библиотеки `LibTooling`, который мы использовали в программе и для которого нужно добавить дополнительную директиву `include`:

```
#include "clang/Tooling/Refactoring.h"
```

Класс `RefactoringTool` координирует выполнение инструментом основных задач, таких как открытие исходных файлов, их парсинг, вызов предикатов AST, вызов обработчиков для выполнения операций при обнаружении соответствий и применение всех модификаций к исходному коду. Именно поэтому после инициализации всех необходимых объектов мы завершили функцию `main` вызовом `RefactoringTool::runAndSave()` – мы передали управление этому классу и позволили ему выполнить все эти задачи.

Далее объявляется объект `MatchFinder` из только что подключенного заголовочного файла. Этот класс выполняет поиск в дереве AST, с критериями которого мы уже определились, воспользовавшись `Clang Query`. Объект `MatchFinder` настраивается предикатами AST и обработчиком, который должен вызываться при обнаружении узла AST, соответствующего критериям поиска. В этом обработчике можно организовать изменение исходного кода. Обработчики реализуются как подклассы класса `MatchCallback`, о котором рассказывается ниже.

Далее объявляются два отдельных объекта обработчиков и связываются с определенными предикатами с помощью

`MatchFinder::addFinder()`. Мы объявили два отдельных обработчика: один для изменения объявлений метода и другой для изменения вызовов метода. Мы назвали эти обработчики `DeclCallback` и `CallCallback`. Поиск будет выполняться с помощью двух составных предикатов AST, опробованных в предыдущем разделе, только вместо имени класса `Animal` мы использовали `ClassName` – имя параметра командной строки, в котором пользователь должен передать имя класса для рефакторинга. Кроме того, мы заменили имя метода `walk` на `OriginalMethodName`, еще один параметр командной строки.

Мы также предусмотрительно использовали новый предикат `id()`, который не влияет на результаты поиска узлов, но присваивает найденному узлу определенное имя. Это поможет обработчикам генерировать замещающие имена. Предикат `id()` принимает два параметра, первый – имя, которое должно использоваться для извлечения узла, и второй – предикат для извлечения именованных узлов.

Первый составной предикат, отвечающий за поиск объявлений членов, присваивает имя узлам `MethodDecl`, а второй, отвечающий за поиск вызовов методов, присваивает имя узлам `CXXMemberExpr`.

Создание обработчиков

При обнаружении искомого узла AST необходимо выполнить некоторые действия. Мы реализовали такую возможность, создав два новых класса, наследующих класс `MatchCallback`, по одному для каждого предиката.

```
class ChangeMemberDecl : public
    ast_matchers::MatchFinder::MatchCallback{
    tooling::Replacements *Replace;
public:
    ChangeMemberDecl(tooling::Replacements *Replace) :
        Replace(Replace) {}
    virtual void run(const ast_matchers::MatchFinder::MatchResult
        &Result) {
        const CXXMethodDecl *method =
            Result.Nodes.getNodeAs<CXXMethodDecl>("methodDecl");
        Replace->insert(Replacement(
            *Result.SourceManager,
            CharSourceRange::getTokenRange(
                SourceRange(method->getLocation()), NewMethodName));
    }
};

class ChangeMemberCall : public
    ast_matchers::MatchFinder::MatchCallback{
```

```

tooling::Replacements *Replace;
public:
ChangeMemberCall(tooling::Replacements *Replace) :
    Replace(Replace) {}
virtual void run(const ast_matchers::MatchFinder::MatchResult
    &Result) {
    const MemberExpr *member =
        Result.Nodes.getNodeAs<MemberExpr>("member");
    Replace->insert(Replacement(
        *Result.SourceManager,
        CharSourceRange::getTokenRange(
            SourceRange(member->getMemberLoc()), NewMethodName));
    }
};

```

Оба класса хранят в своих недрах ссылку на объект типа `Replacements`, который является синонимом для `std::set<Replacement>`. Класс `Replacement` хранит информацию о том, какие строки, в каких файлах и каким текстом следует заменить. Сериализация экземпляров этого класса обсуждалась в описании инструмента `Clang Apply Replacements`. Класс `RefactoringTool` внутренне управляет множеством объектов `Replacement` и именно поэтому мы используем метод `RefactoringTool::getReplacements()` для получения данного множества и инициализации обработчиков в функции `main`.

Мы определили простой конструктор, принимающий указатель на объект `Replacements` и сохраняющий его для дальнейшего использования. Операции, выполняемые обработчиком, реализуются путем переопределения метода `run()`, который оказался на удивление прост. Наш метод принимает объект `MatchResult`. Класс `MatchResult` хранит все узлы, связанные с указанным именем предикатом `id()`.

Эти узлы управляются объектом класса `BoundNodes`, который доступен в объекте `MatchResult` под именем `Nodes`. То есть, первое, что делает метод `run()`, — он получает интересующий нас узел вызовом специализированного метода `BoundNodes::getNodeAs<CXXMethodDecl>`. Как результат, мы получаем ссылку на узел `CXXMethodDecl`, доступную только для чтения.

После получения доступа к узлу, определяющему, как изменить код, нам нужно получить объект `SourceLocation`, определяющий номера строк и столбцов в исходном коде, где находится требуемая лексема. Класс `CXXMethodDecl` наследует суперкласс `Decl`, который представляет обобщенное объявление. Этот обобщенный класс имеет метод `Decl::getLocation()`, возвращающий требуемый объект `SourceLocation`. Имея эту информацию, можно создать первый объ-

ект `Replacement` и вставить его в список изменений, предлагаемых инструментом.

Конструктор `Replacement` требует передачи трех параметров: ссылки на объект `SourceManager`, ссылки на объект `CharSourceRange` и строки с новым текстом замены для фрагмента, местоположение которого определяется двумя первыми параметрами. Класс `SourceManager` – это универсальный компонент Clang, управляющий исходным кодом, загруженным в память. Класс `CharSourceRange` реализует анализ лексемы и определяет ее границы (две точки в файле), тем самым точно определяя, какие символы следует удалить из файла и заменить новым текстом.

Имея эту информацию, мы создаем объект `Replacement` и сохраняем его в множестве, которое управляется объектом `RefactoringTool`, вот и все. О применении или удалении конфликтующих исправлений позаботится сам объект `RefactoringTool`. Не забывайте помещать все локальные объявления в анонимное пространство имен; этот прием помогает избежать ненужного экспортирования символов.

Тестирование нового инструмента рефакторинга

Для тестирования вновь созданного инструмента мы используем симулятор диких животных. Для начала запустите команду `make` и подождите, пока LLVM завершит компиляцию и компоновку нового инструмента. По окончании можете поэкспериментировать с ним, как вам заблагорассудится. Например, проверьте, как выглядят аргументы командной строки, объявленные как объекты `cl::opt`:

```
$ izzyrefactor -help
```

Для работы с инструментом, нам все еще нужно создать базу данных команд компиляции. Чтобы избежать необходимости создавать конфигурационный файл `CMake` и запускать его, мы создали базу данных вручную. Создайте файл с именем `compile_commands.json` и введите в него следующий текст. Замените тег `<FULLPATHTOFILE>` полным путем к каталогу с исходными текстами симулятора:

```
{
{
  "directory": "<FULLPATHTOFILE>",
  "command": "/usr/bin/c++ -o wildlifesim.cpp.o -c <FULLPATHTOFILE~/wildlifesim.cpp",
  "file": "<FULLPATHTOFILE>/wildlifesim.cpp"
```

```
}  
]
```

После сохранения файла базы данных команд компиляции можно приступить к тестированию:

```
$ izzyrefactor -class=Animal -method=walk -newname=run ./wildlifesim.cpp
```

Загляните в исходный файл и убедитесь, что наш инструмент переименовал все определения и вызовы метода `walk`.

На этом мы завершаем наше руководство, но вы можете обратиться к другим источникам информации, перечисленным в следующем разделе, для получения дополнительных знаний.

Дополнительные ресурсы

Дополнительные сведения вы сможете найти по следующим ссылкам:

- <http://clang.llvm.org/docs/HowToSetupToolingForLLVM.html>: содержит дополнительные инструкции по настройке базы данных команд компиляции. После создания этого файла можно настроить текстовый редактор, чтобы он использовал инструмент для проверки исходного кода по требованию.
- <http://clang.llvm.org/docs/Modules.html>: здесь вы найдете дополнительные сведения о реализации поддержки модулей C/C++ в Clang.
- <http://clang.llvm.org/docs/LibASTMatchersTutorial>: еще одно руководство по использованию предикатов AST и LibTooling.
- <http://clang.llvm.org/extra/clang-tidy.html>: содержит инструкцию пользователя для Clang Tidy и многих других инструментов.
- <http://clang.llvm.org/docs/ClangFormat.html>: содержит инструкцию пользователя для ClangFormat.
- <http://www.youtube.com/watch?v=yuIOGfcOH0k>: видеопрезентация Чандлера Каррута (Chandler Carruth) для C++Now, где он рассказывает, как создавать инструменты рефакторинга.

В заключение

В этой главе мы познакомились с инструментами Clang, построенными на основе инфраструктуры LibTooling, упрощающей создание

инструментов для работы с исходным кодом на C/C++. Мы представили следующие инструменты: Clang Tidy, проверяет наличие в исходном коде распространенных нарушений стандартов оформления; Clang Modernizer, автоматически замещает старые приемы программирования на C++ новыми; Clang Apply Replacements, применяет исправления, созданные другими инструментами рефакторинга; Clang-Format, автоматически оформляет отступы и форматирует исходный код на C++; Modularize, упрощает задачу использования поддержки модулей в C++; PPTrace, документирует деятельность препроцессора; и Clang Query, позволяет тестировать предикаты AST. В заключение мы показали, как создать свой собственный инструмент.

Эта глава завершает книгу, но вы не должны останавливаться на достигнутом. В Интернете имеется масса дополнительных материалов о Clang и LLVM в виде руководств и официальной документации. Кроме того, в проекте Clang/LLVM постоянно появляются новые особенности, достойные изучения. Чтобы своевременно узнавать о них, посещайте страницу блога LLVM по адресу: <http://blog.llvm.org>.

Удачной работы!

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- #
- #0, рег 134
- <
- <Целевая_архитектура>GenAsmMatcher.inc 174
- <Целевая_архитектура>GenAsmWriter.inc, файл 174
- <Целевая_архитектура>GenCodeEmitter.inc, файл 174
- <Целевая_архитектура>GenDAGISel.inc, файл 173
- <Целевая_архитектура>GenDisassemblerTables.inc 174
- <Целевая_архитектура>GenInstrInfo.inc, файл 173
- <Целевая_архитектура>InstrFormats.td, файл 170
- <Целевая_архитектура>InstrInfo.td, файл 170
- <Целевая_архитектура>RegisterInfo.td, файл 169
- <Целевая_архитектура>.td, файл 168
- A
- абстрактное синтаксическое дерево (Abstract Syntax Tree, AST)
 - 65, 112
 - узлы 112
- автономные инструменты
 - llc 73
 - lli 73
 - llvm-as 73
 - llvm-dis 73
 - llvm-link 73
 - llvm-mc 73
 - opt 73
- анализатор исходного кода
 - использование для тестирования статического анализатора 265
- архитектурно-зависимые
 - библиотеки
 - <Целевая_архитектура>AsmParser.a 164
 - <Целевая_архитектура>AsmPrinter.a 165
 - <Целевая_архитектура>CodeGen.a 165
 - <Целевая_архитектура>Desc.a 165
 - <Целевая_архитектура>Disassembler.a 165
 - <Целевая_архитектура>Info.a 165
 - архитектурно-зависимые обработчики 197
 - архитектурно-независимые
 - библиотеки
 - AsmParser.a 164
 - AsmPrinter.a 164
 - CodeGen.a 164
 - MC.a 164
 - MCDisassembler.a 164
 - MCJIT.a 164
 - MCParse.a 164
 - SelectionDAG.a 164
 - Target.a 164

Б

- база данных команд компиляции, создание 290
- базовые блоки (Basic Blocks, BB) 135
- библиотеки, Clang
 - libclangAnalysis 98
 - libclangAST 97
 - libclangBasic 98
 - libclangCodeGen 98
 - libclangLex 97
 - libclangParse 97
 - libclangRewrite 98
 - libclangSema 97
- библиотеки LLVM 68
- библиотеки, LLVM
 - libclang 76
 - libclangAnalysis 77
 - libclangDriver 77
 - libLLVMAnalysis 76
 - libLLVMCodeGen 76
 - libLLVMCore 76
 - libLLVMSupport 76
 - libLLVMTarget 76
 - libLLVMX86CodeGen 76
- библиотеки генераторов кода 163
 - зависимые от архитектуры 164
 - независимые от архитектуры 164
- биткод 48
- большие проекты, анализ 269
 - практический пример 271
- быстрый выбор инструкций 185

В

- выбор инструкций 159, 174
 - SelectionDAG, класс 175
- быстрый выбор инструкций 185
- визуализация 184
- легализация DAG 179
- объединение DAG 179
- с преобразованием DAG-to-DAG
 - 181
- упрощение 178

Г

- генератор LLVM IR
 - #include <llvm/ADT/SmallVector.h> 139
 - #include <llvm/Analysis/Verifier.h> 139
 - #include <llvm/Bitcode/ReaderWriter.h> 139
 - #include <llvm/IR/BasicBlock.h> 139
 - #include <llvm/IR/CallingConv.h> 139
 - #include <llvm/IR/Function.h> 139
 - #include <llvm/IR/Instructions.h> 139
 - #include <llvm/IR/LLVMContext.h> 139
 - #include <llvm/IR/Module.h> 139
 - #include <llvm/Support/ToolOutputFile.h> 139
- сборка и запуск 143
- генератор выполняемого кода (backend)
 - библиотеки 163
 - инструменты 161
 - использование TableGen для реализации 165
- обзор 158
- структура 162
- <Целевая_архитектура>GenAsmMatcher.inc, файл 174
- <Целевая_архитектура>GenAsmWriter.inc, файл 174
- <Целевая_архитектура>GenCodeEmitter.inc, файл 174
- <Целевая_архитектура>GenDAGISel.inc, файл 173
- <Целевая_архитектура>GenDisassemblerTables.inc, файл 174
- <Целевая_архитектура>GenInstrInfo.inc, файл 173

<Целевая_архитектура>InstrFormats.td, файл 170

<Целевая_архитектура>InstrInfo.td, файл 170

<Целевая_архитектура>.td, файл 168

генератор кода C++ 144

граф достижимых состояний программы 263

граф потока выполнения 263

граф управляющей логики (Control Flow Graph, CFG) 118

Д

динамическая компиляция (Just-In-Time Compilation) 31

динамическая компоновка 226

динамические разделяемые объекты (Dynamic Shared Object, DSO) 211

динамический компилятор 208

другие ресурсы 233

инструменты 231

механизм выполнения 210

обзор 209

преимущества 208

управление памятью 212

диспетчер памяти 227

диспетчер проходов 83

доминаторные деревья 149

драйвер

- использование для тестирования статического анализатора 265

драйвер компилятора

- взаимодействие с 71
- описание 71

З

замена виртуальных регистров 196

значения по умолчанию параметров

- BUILD_SHARED_LIBS 34
- CMAKE_BUILD_TYPE 34

- CMAKE_ENABLE_ASSERTIONS 34
- CMAKE_INSTALL_PREFIX 34
- LLVM_TARGETS_TO_BUILD 34

И

инструкции MC 200

инструментарий компиляции

- анализатор исходного кода 242
- асемблер 242
- библиотеки времени выполнения 241
- компоновщик 242
- подготовка 240
- стандартные библиотеки C и C++ 240

инструменты генераторов кода 161

инструменты рефакторинга 294

информация о целевой архитектуре 215

инфраструктура, LLVM 67

- анализатор исходного кода 68
- генератор выполняемого кода 69

инфраструктура машинного кода 199

К

канадский крест 239

классический механизм предупреждений

- и статический анализатор Clang 258

компилятор, взаимодействия

- через структуры в памяти 69
- через файлы 69

компилятор на основе LLVM 67

компиляция, DragonEgg

- с инструментами LLVM 54

кросс-компиляция

- одноплатные компьютеры 254
- тестирование 254

кросс-компиляция с аргументами командной строки Clang 244

зависимости 245

изменение корневого каталога 248

параметры драйвера,
 определяющие архитектуру 244
 потенциальные проблемы 247
 установка GCC 247
 кросс-платформенная компиляция 235

Л

легализация DAG 179
 легковесные ссылки на строки 81
 лексический анализ
 использование libclang 107
 препроцессор 109
 проверка лексических ошибок 107
 лексический анализ 105
 ленивая, или отложенная
 компиляция 211

М

маршруты инструкций 186
 машинные инструкции 188
 машинный код (MC) 199
 международный конкурс
 запутывания кода на C (International Obfuscated C Code Contest, IOCCC) 110
 механизм выполнения 210
 внешние глобальные переменные,
 компиляция 211
 ленивая, или отложенная
 компиляция 211
 механизм символического
 выполнения 259
 возможности 262
 механизмы вывода целевого кода 214
 модули
 компиляция модулей 224

Н

неявные зависимости 149

О

объединение DAG 179

одноплатные компьютеры 254

Beagle Board 254
 Carambola-2 254
 Panda Board 254
 SEAD-2 254

операции анализатора исходного
 кода

ASTView 96
 Clang 96
 EmitBC 96
 EmitObj 96
 FixIt 96
 PluginAction 96
 RunAnalysis 96

определение опасностей 188

оптимизация

на уровне IR 145
 API проходов 151
 времени компоновки 145
 зависимости между проходами 149
 межпроцедурная оптимизация 145
 проходы 146

ориентированный ациклический
 граф (Directed Acyclic Graph,
 DAG) 65, 159

П

пакет тестов LLVM 56
 URL 56
 перенастраиваемые компиляторы 128
 планирование инструкций 160, 186
 маршруты инструкций 186
 подклассы Pass
 BasicBlockPass, класс 151
 URL документации 152
 позиционно-независимый
 программный код (Position
 Independent Code, PIC) 42
 предварительно скомпилированные
 заголовочные файлы (Pre-
 compiled Headers, PCH) 103
 предикаты AST 321

- включение в программный код 324
 - комбинирование 322
 - преобразования, Clang Modernizer
 - преобразование передачи аргументов по значению 295
 - преобразование пустых указателей 295
 - преобразование с добавлением ключевого слова `auto` 295
 - преобразование с добавлением ключевого слова `override` 295
 - преобразование с заменой `auto_ptr` 295
 - преобразование циклов 295
 - приемы программирования на C++ 78
 - легковесные ссылки на строки 81
 - полиморфизм 78
 - расширяемый интерфейс проходов 83
 - шаблоны 79
 - эффективные 80
 - прикладной двоичный интерфейс (Application Binary Interface, ABI) 235
 - программный интерфейс на C/C++ 301
 - задача компоновщика 301
 - модули 304
 - на стороне анализатора исходного кода 303
 - проблема препроцессора C/C++ 303
 - пролог и эпилог функций 198
 - прохода для генератора кода, реализация 203
- Р**
- распределение регистров 160, 189
 - архитектурно-зависимые обработчики 197
 - замена виртуальных регистров 196
 - обзор 189
 - объединение 191
- расширяемый интерфейс проходов 83
 - реализация собственного прохода 152
 - сборка и запуск 153, 155
- С**
- семантический анализ 119
 - проверка семантических ошибок 120
 - создание кода промежуточного представления 121
 - синтаксический анализ 112
 - исследование с помощью отладчика 114
 - проверка синтаксических ошибок 115
 - сериализация AST 119
 - скомпилированные пакеты 25
 - загрузка с официального сайта 25
 - установка LLVM 25
 - собственного прохода для генератора кода 208, 235, 257, 290
 - собственный генератор LLVM IR 139
 - собственный инструмент
 - создание 314
 - определение задачи 315
 - структура каталогов с исходным кодом 315
 - создание обработчиков 326
 - тестирование 328
 - шаблонный код 317
 - состояния модулей
 - готовы 223
 - добавлен 223
 - загружен 223
 - средства проверки 257
 - макрос регистрации 282
 - подкласс Checker, определение 280
 - подкласс Checker, реализация 283
 - сборка 286
 - собственные 275
 - класс состояния 279
 - реализация 277
 - тестирование 286
 - функция регистрации 286

статический анализатор 258
 статический анализатор Clang
 анализ больших проектов 269
 практический пример 271
 архитектура проекта 275
 графические отчеты в HTML 269
 документация
 URL 287
 и классический механизм
 предупреждений 258
 использование в Xcode IDE 268
 расширение 275, 277
 тестирование 265
 тестирование, с использованием
 драйвера 265
 структура, генератора кода 162

Т

триады определения целевой
 архитектуры 238

У

управление памятью 212
 упрощение, выбор инструкций 178
 установка, DragonEgg 53
 установка, LLVM
 загрузка исходных текстов 29
 сборка в Mac OS X и Xcode 41
 сборка в Unix с использованием
 других подходов 36
 сборка в Windows 37
 сборка из исходных текстов 28
 системные требования 28
 сборка и установка в Unix 32
 сборка с использованием
 CMake 33
 сборка с использованием Ninja 33
 сборка с использованием сценария
 configure 30
 с использованием диспетчера
 пакетов 27
 скомпилированных пакетов 25

Э

экспоненциальная временная
 сложность 258
 эмиссия кода 160, 200
 эпилог и пролог функций 198
 этапы генератора кода 159
 выбор инструкций 159
 планирование инструкций 160
 распределение регистров 160
 эмиссия кода 160

Я

явные зависимости 149
 ядро LLVM 68

А

ABI (Application Binary Interface
 прикладной двоичный
 интерфейс) 235
 Abstract Syntax Tree (AST),
 абстрактное синтаксическое
 дерево 65
 -analyzer-checker-help, флаг 266
 --analyze, флаг 266
 analyzer-checker, флаг 268
 API проходов 151
 реализация собственного прохода
 152
 сборка и запуск 153, 155
 AST
 узлы 112

В

BuildMI(), метод 198

С

Clang
 Clang Check 48
 Clang Format 48

- Compiler-RT 50
 - Modularize 49
 - URL 94
 - инструменты 295
 - описание 48
 - этапы работы анализатора 105
 - лексический анализ 105
 - использование libclang 107
 - препроцессор 109
 - проверка лексических ошибок 107
 - семантический анализ 119
 - синтаксический анализ 112
 - Clang Apply Replacements, инструмент 296
 - clang -cc1, инструмент 95
 - Clang Check 48
 - Clang Check, инструмент 313
 - Clang Format 48
 - ClangFormat, инструмент 298
 - Clang Front End Developer List URL 90
 - Clang Modernizer 48
 - Clang Modernizer, инструмент 295
 - Clang Query, инструмент 311
 - Clang Static Analyzer 258
 - clang-tidy, инструмент 292, 293
 - Clang Tidy, инструмент 49
 - clang-tools-extra, репозиторий 48
 - URL 49
 - установка 49
 - clang_visitChildren(), функция 117
 - Clang, диагностика 100
 - чтение 101
 - Clang, инструменты
 - Clang Apply Replacements 296
 - Clang Check 313
 - ClangFormat 298
 - Clang Modernizer 295
 - Clang Query 311
 - Clang-tidy 292
 - Clang-tidy, инструмент 293
 - Modularize 300
 - Module Map Checker 308
 - PPTrace 309
 - remove-cstr-calls 314
 - рефакторинга 294
 - Clang, кросс-компилятор
 - создание 250
 - альтернативные методы сборки 252
 - ELLCC 253
 - EmbToolkit 253
 - Ninja 253
 - параметры настройки 250
 - enable-targets 250
 - target 250
 - with-c-include-dirs 251
 - with-default-sysroot 251
 - with-gcc-toolchain 251
 - сборка и установка 251
 - Clang, проект
 - библиотеки 97
 - операции 96
 - описание 94
 - CMake
 - использование для сборки LLVM 33
 - устранение ошибок сборки 36
 - CodeGen, каталог 162
 - Compiler-RT
 - URL для загрузки 50
 - в действии 51
 - описание 50
 - configure, сценарий
 - enable-assertions, параметр 31
 - enable-jit, параметр 31
 - enable-optimized, параметр 31
 - enable-shared, параметр 31
 - enable-targets, параметр 31
 - prefix, параметр 31
 - copyPhysReg(), метод 198
 - Cross Linux from Scratch, руководство 246
- ## D
- DAG (Directed Acyclic Graph

ориентированный
 ациклический граф) 159
 DAG-to-DAG, преобразование 181
 сопоставление с шаблоном 182
 Directed Acyclic Graph (DAG),
 ориентированный
 ациклический граф 65
 DragonEgg 52
 LLDB, использование 57
 пример сеанса отладки 58
 URL для загрузки 53
 использование 54
 компиляция с инструментами
 LLVM 54
 пакет тестов LLVM 56
 сборка 53
 установка 53

Е

ELLCC 253
 EmbToolkit 253

F

FTL, компонент 210

G

GCC
 в сравнении с LLVM 236
 getGlobalContext(), функция 140
 getPointerToFunction(), метод 221
 Git-зеркала репозитория, получение
 исходных текстов 30

Н

HTML
 графические отчеты 269

I

InitializeNativeTarget(), метод 218
 IR
 описание 64

IR, форматы
 инструменты для работы с 131
 isLoadFromStackSlot(), метод 198
 isStoreToStackSlot(), метод 198

J

JITCodeEmitter, класс 213
 JITMemoryManager, класс
 использование 214
 JIT, класс 212
 использование 217
 обобщенные значения 221
 JIT-компилятор 208
 другие ресурсы 233
 инструменты 231
 механизм выполнения 210
 обзор 209
 преимущества 208
 управление памятью 212

L

libc++
 URL 241
 libclang
 URL 97
 использование 97, 98
 libclangAnalysis, библиотека 77, 98
 libclangAST, библиотека 97
 libclangBasic, библиотека 98
 libclangCodeGen, библиотека 98
 libclangDriver, библиотека 77
 libclangLex, библиотека 97
 libclangParse, библиотека 97
 libclangRewrite, библиотека 98
 libclangSema, библиотека 97
 libclang, библиотека 76
 libc++, стандартная библиотека 59
 URL 61
 libLLVMAnalysis, библиотека 76
 libLLVMCodeGen, библиотека 76
 libLLVMCore, библиотека 76
 libLLVMSupport, библиотека 76

- libLLVMTarget, библиотека 76
- libLLVMX86CodeGen, библиотека 76
- LibTooling 290
- lс, инструмент 73
 - использование 161
- lld
 - URL 242
- LLDB 57
 - URL 57
 - использование 57
 - пример сеанса отладки 58
- lli, инструмент 73, 231
- LLVM
 - анализатор исходного кода 68
 - библиотеки 68, 76
 - внутренняя организация 75
 - в сравнении с GCC 236
 - генератор выполняемого кода 69
 - загрузка исходных текстов 29
 - инфраструктура 67
 - исходный код как документация 89
 - нумерация версий 24
 - обращение к сообществу за помощью 90
 - пакет тестов 56
 - поддерживаемый список платформ 28
 - приемы программирования на C++ 78
 - легковесные ссылки на строки 81
 - полиморфизм 78
 - расширяемый интерфейс проходов 83
 - шаблоны 79
 - эффективные 80
 - принципы организации 64
 - сборка в Mac OS X и Xcode 41
 - сборка в Unix с использованием других подходов 36
 - сборка в Windows 37
 - сборка из исходных текстов 28
 - системные требования 28
 - сборка и установка в Unix 32
 - сборка с использованием CMake 33
 - сборка с использованием Ninja 33
 - сборка с использованием сценария configure 30
 - советы по навигации в исходных текстах 89
 - установка с использованием диспетчера пакетов 27
 - установка скомпилированных пакетов 25
 - чтение журнала изменений SVN 90
 - ядро 68
- llvm-as, инструмент 73
- LLVM core Developer List
 - URL 90
- LLVMdev
 - URL 66
- llvm-dis, инструмент 73
- llvm-extract, инструмент 131
- LLVM IR
 - биткод 131
 - зависимость от целевой архитектуры 130
 - обзор 127
 - описание 64
 - оптимизация 145
 - API проходов 151
 - времени компоновки 145
 - зависимости между проходами 149
 - межпроцедурная 145
 - проходы 146
 - представление в памяти 136
 - BasicBlock, класс 137
 - Function, класс 137
 - Instruction, класс 137
 - Module, класс 137
 - User, класс 138
 - Value, класс 138
 - реализация собственного генератора 139
 - синтаксис языка 132

- фундаментальные свойства 133
 - llvm::JIT, инфраструктура
 - введение 213
 - запись двоичного кода в память 213
 - информация о целевой архитектуре 215
 - механизмы вывода целевого кода 214
 - llvm-link, инструмент 73
 - llvm::MCJIT, инфраструктура 222
 - MCJIT, механизм 222, 223
 - MCJIT, механизм, динамическая компоновка 226
 - MCJIT, механизм, диспетчер памяти 227
 - MCJIT, механизм, использование 228
 - MCJIT, механизм, компиляция модулей 224
 - MCJIT, механизм, окончательная подготовка модуля 228
 - llvm-mc, инструмент 73
 - llvm-rtldyld, инструмент 232
 - LLVM, проект 128
 - инфраструктура 67
 - loadRegFromStackSlot(), метод 198
- М**
- m32, флаг 51
 - MachineCodeEmitter, класс 213
 - метод allocateSpace() 213
 - метод emitAlignment() 213
 - метод emitByte() 213
 - метод emitWordBE() 213
 - метод emitWordLE() 213
 - Makefile
 - создание 85
 - makeLLVMModule(), функция 140
 - MCJIT, класс 212
 - MCJIT, механизм 222, 223
 - динамическая компоновка 226
 - диспетчер памяти 227
 - использование 228
 - компиляция модулей 224
 - окончательная подготовка модуля 228
 - состояния модулей 223
 - MemoryBuffer, класс 225
 - Modularize, инструмент 300
 - Module Map Checker, инструмент 308
 - multilib 243
 - структура каталогов 243
- N**
- Ninja 253
- O**
- ObjectBufferStream, класс 224
 - ObjectBuffer, класс 224
 - ObjectFile, класс 225
 - ObjectImage, класс 225
 - opt, инструмент 73, 150
- P**
- placePlaceNameподклассы
 - PlaceTypePass
 - FunctionPass, класс 151
 - ModulePass, класс 151
 - PPTrace 49
 - PPTrace, инструмент 309
- R**
- remove-cstr-calls, инструмент 314
 - RTDyldMemoryManager, класс 212
 - метод allocateCodeSection() 212
 - метод allocateDataSection() 212
 - метод finalizeMemory() 212
 - метод getSymbolAddress() 212
 - runFunction(), метод 221
 - RuntimeDyld, динамический компоновщик 226
- S**
- scan-build, инструмент 270
 - SCRAM(), функция 277

SelectionDAG, класс 175
Select(), метод 182
setDataLayout(), функция 140
setTargetTriple(), функция 140
Static Single Assignment (SSA),
 форма 133
storeRegToStackSlot(), метод 198
SVN, репозиторий
 получение исходных текстов 29
syntax-only, флаг 268

T

TableGen
 описание 100
TableGen, каталог 162
TableGen, язык 165
target datalayout, конструкция 133
TargetJITInfo::emitFunctionStub(),
 метод 216

TargetJITInfo::relocate(), метод 216
TargetJITInfo::replaceMachineCodeFor
 Function(), метод 216
TargetJITInfo, класс 215
TargetPassConfig, класс 203
TargetRegisterInfo, класс 198
Target, каталог 162
turnReactorOn(), функция 277

V

ViewVC
 URL 91

X

-Xanalyzer, флаг 266
Xcode IDE
 использование статического
 анализатора Clang 268

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, пр. Андропова, д. 38.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru.**

Оптовые закупки: тел. **+7 (499) 782-38-89.**

Электронный адрес: **books@aliants-kniga.ru.**

Бруно Кардос Лопес (Bruno Cardoso Lopes)
Рафаэль Аулер (Rafael Auler)

LLVM: инфраструктура для разработки компиляторов

Знакомство с основами LLVM и использование базовых библиотек для создания продвинутых инструментов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Парапская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 22,79.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru

LLVM — новейший фреймворк для разработки компиляторов. Благодаря простоте расширения и организации в виде множества библиотек, LLVM легко поддается освоению даже начинающими программистами, вопреки устоявшемуся мнению о сложности разработки компиляторов.

Сначала эта книга покажет, как настроить, собрать и установить библиотеки, инструменты и внешние проекты LLVM. Затем вы познакомитесь с архитектурой LLVM и особенностями работы всех компонентов компилятора: анализатора исходных текстов, генератора кода промежуточного представления, генератора выполняемого кода, механизма JIT-компиляции, возможностями кросс-компиляции и интерфейсом расширений. На множестве наглядных примеров и фрагментов исходного кода эта книга поможет вам войти в мир разработки компиляторов на основе LLVM.

- Настройка, сборка и установка дополнительных открытых проектов LLVM, включая инструменты Clang, статический анализатор, Compiler-RT, LLDB, LLDB, DragonEgg, libc++ и комплект тестов для LLVM
- Архитектура библиотек LLVM и особенности взаимодействий между библиотеками и автономными инструментами
- Стадии обработки исходного программного кода и порядок выполнения лексического, синтаксического и семантического анализа анализатором исходного кода Clang
- Как создаются и обрабатываются файлы LLVM IR с промежуточным представлением, а также разработка собственных проходов анализа и трансформации IR-кода
- Создание инструментов для использования средств динамической компиляции LLVM (Just-in-Time, JIT)
- Поиск ошибок и оптимизация кода с помощью статического анализатора
- Анализ архитектуры исходного кода и создание инструментов его преобразования с использованием LibClang, LibTooling и интерфейса расширений Clang



Эта книга адресована энтузиастам, студентам, изучающим информационные технологии, и разработчикам компиляторов, интересующимся фреймворком LLVM. Читатели должны знать язык программирования C++ и, желательно, некоторые представления о теории компиляции. И для начинающих, и для опытных специалистов эта книга послужит практическим введением в LLVM, не содержащим сложных сценариев. Если вас интересует данная технология, тогда эта книга определено для вас.

ДМК
ИЗДАТЕЛЬСТВО
www.dmk.pf

Интернет-магазин www.dmkpress.com

Книга-почтой: orders@aliants-kniga.ru

Оптовая продажа: «Альянс-книга»
(499)782-3889, books@aliants-kniga.ru

ISBN 978-5-97060-305-5



9 785970 603055 >