УДК 519.683.8

Подходы к кодогенерации.

Старичков Н.Ю.
Московский физико-технический институт(ГУ)
Фирма «1С»

Современные приложения часто должны выполнять разнообразные пользовательские запросы. Это могут быть веб-приложения, базы данных, различные системы учета и планирования, системы математических вычислений. Стандартный подход в реализации такого функционала, помимо разбора самого текста запроса и т.д. обычно сводился к вызову разнообразных функций, их параметризацией и т.д. С появлением C++ стала возможна реализация подобного функционала с помощью шаблонов. Однако и тот, и другой подход приводят к тому, что пользовательские запросы выполняются намного медленнее тех, которые заложены в систему при ее написании. Причин этому несколько. Для примера рассмотрим систему для математических вычислений, в которой пользователь может создавать свои типы и структуры данных. Тогда любая операция, которую пользователь захочет произвести над вновь созданным типом данных, будет требовать вызова функции, выполняющей эту операцию. Пусть, к примеру, пользователь реализовал матрицы. Тогда сложение матриц будет требовать вызова отдельной функции. Но если нужно сложить два массива матриц поэлементно? Тогда для каждой пары элементов будет вызвана функция сложения матриц. Помимо очевидного минуса непосредственно во множестве лишних вызовов функций, есть другой, не менее значимый - при множественных вызовах функций эффективность кэширования данных процессором существенно снижается(так называемые кэш-миссы), что также приводит к падению производительности. Однако, с развитием технологий, стали появляться инструменты, позволяющие выполнять JIT-компиляцию, т.е. компиляцию исходного кода «на лету». Это позволило разработать новый подход к выполнению пользовательских запросов: вместо того, чтобы комбинировать вызовы неких функций для его выполнения, можно сгенерировать код, который выполнит этот запрос, скомпилировать его, и эффективно выполнить, без лишних вызовов функций и с меньшей долей кэш-миссов. Однако, генерация кода и его компиляция тоже требуют времени, поэтому такой подход показывает свою эффективность начиная с достаточно объемных запросов. Подчеркнем, что оверхед для стандартных подходов увеличивается с runtime-объемом запроса(например, увеличение количества обрабатываемых элементов и т.д.), а оверхед для кодогенерации постоянен, и от runtime-параметров запроса не зависит.

Можно предложить различные подходы для кодогенерации. Во-первых, существующие технологии предоставляют множество вариантов для выбора языка, код на котором будет генерироваться. Рассмотрим различные варианты. Первый класс вариантов - языки высокого уровня - предоставляют большой объем выразительных средств, и сгенерированный на них код получается компактным и легко читаемым человеком. Плюсами данного подхода являются легкость генерации, простота кодирования системы кодогенерации, легкая

отладка как кодогенератора, так и сгенерированного им кода. Ключевым минусом является очень большое время компиляции. Его можно несколько сократить, если использовать компиляцию без оптимизаций, но тогда и прирост производительности от использования генерации кода тоже падает. Второй класс - языки низкого уровня. Их главные плюсы - маленькое время компиляции и возможности закладки оптимизаций в процессе кодогенерации(т.к. в отличие от компилятора, кодогенератор знает особенности прикладной задачи). Минусом стоит отметить большой объем генерируемого кода, затрудненную отладку кодогенератора и очень непростую отладку сгенерированного кода. Сам по себе сгенерированный код достаточно трудночитаем человеком.

Однако, т.к. мы боремся в первую очередь за скорость работы, наш выбор - языки низкого уровня. Самый распространенный из них - ассемблер, к сожалению, нам не подходит, потому что под разные архитектуры и разные процессоры существуют различные его диалекты, и универсальный кодогенератор сделать не получится. Однако, проект LLVM в своем составе содержит язык программирования IR(intermediate representation), который отлично подходит для задач кодогенерации. Во-первых, самой своей структурой(об этом ниже), во-вторых, поддержкой его компиляции на различных платформах(предоставляемых инфраструктурой LLVM), и, в-третьих, отличной документацией. Также стоит отметить, что в LLVM уже имеется встроенный кодогенератор, который, однако, не слишком удобен для решения прикладных задач, а скорее подходит для задач написания собственных компиляторов.

Теперь уточним основные особенности языка IR LLVM. Во-первых, типы данных. IR - язык со строгой типизацией. Во всех операциях, инструкциях, объявлениях функций и т.д. явно указывается тип входящих в них выражений. IR позволяет создавать переменные целочисленных типов любой размерности от одного бита. Верхний предел неограничен, но в документации явно указывается, что при компиляции кода, содержащего переменные большей размерности, чем поддерживает машина, эти переменные будут преобразованы. Объявление таких типов выглядит как iX, где X - число(например, i8 - восьмибитное целое). Различий между знаковыми и беззнаковыми целыми в типах переменных нет(это проявляется только при использовании некоторых инструкций). Также есть типы double и float. Также возможно создание собственных структур данных, массивов, и массивов из структур данных. Имеются векторы для эффективной реализации SIMD-операций(размер вектора - обязательно одна из степеней 2). (SIMD - Single Instruction, Multiply Data - принцип вычислений, позволяющий реализовывать параллельность на уровне данных). Конечно же, имеются указатели. Система типов в IR рекурсивна, т.е. можно, например, создать массив указателей. Одна из ключевых особенностей IR LLVM - SSA. SSA - Single Assignment Form - форма представления кода, в которой любое значение присваивается переменной только один раз. Например, нельзя написать %z = sum i32 %x, %y; %z = sum i32 %x, 5. Это накладывает определенные ограничения на стиль программирования на IR. Практически все операции производятся с указателями, переменные используются только в качестве промежуточных или вспомогательных полей, и ограничены в использовании сравнительно небольшим участком кода. Другое дело - указатели. Фактически, это единственный эффективный способ передачи данных между участками кода (а для передачи множества переменных разом - вообще единственный, если не вводить дополнительные структуры данных). Неразрывно с SSA связана еще одна особенность IR LLVM - phi-инструкция.

Объясним подробнее, что это такое. В IR для логической разметки кода существуют так называемые метки(label). Поток управления может идти по коду последовательно, по очереди выполняя код в каждом блоке, а может «прыгать» по меткам с помощью инструкции br. Инструкция br имеет две реализации - для безусловных и условных переходов. Безусловный переход переводит поток управления на указанную в теле инструкции метку, условный переход переводит поток управления на одну из двух указанных меток в зависимости от значения переменной, переданной в инструкцию br. Это позволяет выстраивать логику на «выходе» из блока. Однако, часто может потребоваться устроить некую логику на «входе» в блок. Знание из какого именно блока был осуществлен переход в текущий блок не так важно, а вот получение каких-то данных из предыдущих блоков требуется практически всегда. Для этого и нужна инструкция phi. Она возвращает в качестве результата значение одной из переменных, указанных в формате имя переменной - имя метки, в зависимости от того, из какого блока(находящегося под меткой с определенным именем) был осуществлен переход в текущий блок. Например, если в текущий блок C может быть осуществлен переход из блоков A и B, и в коде есть подобная строка: res = phi i32 [%a, %A], [%b, %B], то это означает, что в переменную res в результате выполнения этой инструкции будет загружено значение переменной a, если переход в текущий блок был осуществлен из блока A, или переменной b, если переход был из блока B. Это позволяет выстаивать логику выполнения программы через переходы(напомним, что в IR нет высокоуровневых конструкций наподобие условных операторов, операторов множественного выбора, циклов и т.д.).

Следующий важный вопрос - как устроить логику кодогенерации. Понятно, что для поддержки системой какого-то достаточно большого множества возможных пользовательских запросов необходимо реализовать генерацию отдельных секций(кусков) IR кода. Опустим здесь вполне известные проблемы и способы их решения по парсингу пользовательских запросов, построению синтаксического дерева, дерева логических и физических операторов. Будем считать, что на вход кодогенератора пользовательский запрос попадает в формате дерева физических операторов. Чтобы множество поддерживаемых запросов было достаточно широко, и нам не пришлось реализовывать большое количество различных функций по генерации секций, необходимо секции делать очень небольшими, такими, чтобы они могли быть использованы в как можно большем количестве запросов. Будем предполагать, что и дерево физических операторов детализировано до уровня, что в нодах(вершинах) находятся именно такие секции. Другими словами, необходимо определить базис секций кодогенерации. Отметим, что он должен обладать некоторыми свойствами, присущими любому базису. Базис должен быть минимальным(т.е. элементы в нем должны быть независимы - невозможно реализовать один из них с использованием других) - это необходимо для минимального объема работ по реализации кодогенератора, а так же более эффективного кеширования и переиспользования уже сгенерированного кода. Очевидно, что базис должен обладать свойством полноты в том множестве пользовательских запросов, которые предполагается поддержать (т.е. чтобы любой запрос, который может поступить на вход, был реализуем с помощью элементов базиса). Также стоит постараться сделать свой выбор так, чтобы даже относительно большие, но очень часто используемые конструкции попали в базис (опять же, дело, в первую

очередь, в возможности эффективного кеширования). Сам выбор базиса зависит, в наибольшей степени, от прикладной области и от того множества запросов, которые необходимо поддержать. Также стоит обратить внимание, что внесение некоторых функций в базис позволит сделать кодогенерацию некоторого подмножества пользовательских вопросов более быстрым и эффективным, что важно, если среди множества пользовательсих запросов можно выделить подмножество наиболее популярных и/или наиболее требовательных к скорости выполнения.

Следующая проблема, которую предстоит решить, это выбор стратегии того, как генерируемые базовые секции будут связываться друг с другом. Однако, сначала нужно определиться как решить следующую проблему - т.к. в IR используется SSA, необходимо обеспечить различные имена всех переменных в коде, сгенерированном для выполнения пользовательского запроса. Так как различные секции генерируются отдельно, необходимо разработать механизм, обеспечивающий контроль и обеспечение свойства, что в этих секциях не будет переменных с одинаковыми именами. То же требование относится и к именам меток. Можно предложить две противоположные концепции в реализации такого механизма. Первый из них заключается в том, что в первоначально сгенерированном коде отсутствуют реальные имена переменных, а присутствует лишь некое условное число. Это число выдается функциям генерации кода по запросу к специальному классу, внутри которого, упрощенно, находятся счетчики текущего номера переменной и номера метки. Такой подход поддерживается непосредственно инфраструктурой LLVM, и, при соблюдении определенных несложных синтаксических правил, такой «первичный» код будет корректно воспринят компилятором. Тем самым мы можем «переложить» проблему именования переменных и меток на инфраструктуру LLVM. Однако, такой подход имеет несколько очевидных минусов. Во-первых, существенно усложняется непосредственно генерация кода. Да, создание переменной не вызывает проблем, но, например, присваивание этой переменной значения суммы двух предыдущих уже вызывает проблемы - необходимо хранить, например, стек имен(номеров) предыдущих переменных, чтобы корректно сгенерировать требуемую инструкцию. Также, как минус данного подхода, стоит отметить, что параллельная генерация секций становится невозможной(или слишком медленный) - все функции генерации будут упираться в бутылочное горлышко в виде класса, выдающего номера переменных и меток, которые в данный момент можно использовать. Все из-за того же SSA, практически каждая новая инструкция в IR LLVM требует создания новой переменной, следовательно, обращений к классу за очередным именем будет очень много. Да, можно предложить способы обхода этой проблемы - например, сделать несколько экземпляров такого классов. Уникальность будет обеспечиваться каким-то правилом, наложенным на числа, выдаваемым для генераторов(например, один экземпляр выдает последовательно числа, начиная с 0, а второй - начиная с 1000000, теоретически это, конечно, не является решением проблемы, но в реальных задачах и в реальных пользовательских запросах число переменных всегда можно легко оценить сверху и выбрать необходимое для отсутствия конфликтов начальное число для второго экземпляра). Но подобное решение сразу перечеркивает один из главных плюсов этого подхода в целом - т.к. LLVM требует, чтобы условные числовые имена переменных и меток были последовательны, мы, обеспечив возможность параллелизации, будем

вынуждены самостоятельно писать механизм дальнейшего переименования переменных и меток в получившимся коде. Третьим минусом стоит отметить то, что мы лишаемся возможности кешировать уже сгенерированные секции, опять же по той причине, что повторное переиспользование части из них с вновь сгенерированными будет вызывать нарушения последовательности числовых условных имен переменных и меток, т.е. снова использование инфраструктуры LLVM станет невозможно. Этот подход удобен лишь, если использовать всю инфраструктуру LLVM в целом - т.е. использовать встроенный в нее кодогенератор. Альтернативным подходом является механизм написания кодогенератора таким образом, чтобы в каждой сгенерированной секции были свои уникальные имена переменных изначально. Одним из самых простых способов может являться добавление имени секции в качестве префикса или суффикса имен переменных и меток. Однако, в одном пользовательском запросе одна и та же базисная секция может использоваться несколько раз, и нам необходимо обеспечить механизм выдачи каких-то дополнительных префиксов или суффиксов именам переменных и меток в этих секциях для обеспечения уникальности в разных экземплярах одной и той же секции. Самым простым решением является обязательная передача некоего id, который передается в функцию генерации каждой секции. И этот id добавляется в имя каждой переменной и каждой метки внутри данной секции. За выдачу уникальных id отвечает некий класс. Однако, это не является барьером к параллелизации генерации секций, т.к. обращение за id происходит лишь один раз за весь процесс генерации секции. Также плюсом данного решения является то, что внутри секций мы можем использовать «человеческие» имена переменных и меток, что упрощает написание кодогенератора, уменьшает вероятность допустить ошибку, а также упрощает анализ и отладку сгенерированного кода.

Теперь нужно разработать механизм, который позволит логически связывать сгенерированные секции. Понятно, что механизм должен быть удобен. Т.к. на вход кодогенератора попадает дерево физических операторов, мы видим какие секции с какими должны быть связаны уже в момент генерации кода. Соответственно, первый подход заключается в «связывании» секций между собой в момент генерации. Однако, конкретная базовая секция в различных вариантах пользовательских запросов, в различных местах одного запроса может связываться с различными базовыми секциями. Соответственно, требуется зафиксировать формат взаимодействия секций. Т.к. секции - это не функции, у них нет механизма аргументов и возвращаемого значения, необходимо зафиксировать конкретные имена переменных, которые в рамках данной секции будут выполнять роль переменных и возвращаемого значения/значений. В таком случае необходимо ввести новую базовую секцию - адаптер, которая будет фактически «переименовывать» переменные. Т.к. порядок следований секций и взаимодействия может меняться, невозможно «подогнать» имя возвращаемого значения одной секции к конкретному имени аргумента последующей секции, тем самым избавившись от адаптера. Стоит отметить, что появление адаптера практически не влияет на эффективность выполнения исходного кода, т.к. практически все используемые и передаваемые между секциями переменные являются указателями, т.е. копирования данных в адаптере происходить не будет. Однако, если учесть выбранный нами способ реализации механизма обеспечения уникальных имен переменных и меток, в адаптер придется передавать не только имена переменных из предшествующей

и последующей секции, но и их id. Это, однако, неудобно. Когда мы генерируем код секций, нам очень удобно перемещаться по дереву физических операторов в каком-то порядке (например, post-order), а такой подход заставит нас помимо движения в порядке обхода делать шаги назад (rollback), что усложняет код обхода дерева и вызова необходимых функций кодогенерации. Другой подход к логическому связыванию секций заключается в использовании условных имен для переменных-аргументов и переменных-возвращаемых значений. Т.е. в процессе генерации секций мы используем некие специальные имена, которые позже, в процессе связывания будут заменены на настоящие. Предлагается использовать имена наподобие %_gX, где X - число. Во время генерации в коде используются имена %_gX, а на втором этапе, в момент объединения секций в единый блок кода эти имена заменяются на указанные в параметрах линковки секций. При таком подходе разница между переменными-аргументами и переменными-возвращаемыми значениями практически отсутствует.

Теперь, когда мы определились со связыванием переменных между секциями, нужно разработать механизм логического связывания секций.

Для передачи потока управления от секции к секции предлагается использовать безусловные переходы(unconditional jumps, инструкция br без указания переменной). Это позволит выстраивать линейные программы. Однако, в прикладных задачах этого недостаточно: например, при возникновении ошибки мы можем захотеть перейти в секцию, которая сохранит код ошибки в файл, а в случае безошибочного выполнения - перейти к следующему шагу исполнения. То есть механизм безусловных переходов позволяет подставить «следующей» за данной любую секцию, но если мы не предоставим возможности условного перехода, т.е. перехода в одну из двух секций в зависимости от значения переменной(conditional jumps), наш механизм будет неудобным. Да, можно обойтись и без условных переходов, оставив подобные ветвления лишь внутри секций. Но это очень сильно усложнит секции, резко увеличит число базовых секций и не позволит реализовать эффективное кэширование.

Т.к. в момент генерации секции неизвестно, какая именно секция или секции будут идти вслед за ней, предлагается реализовать механизм переходов аналогично механизму переменных, т.е. использовать условные имена для меток последующих секций. Соответственно, предлагается для безусловных переходов использовать условные имена %_uX, где X - число, а для безусловных - %_cX, где X - число. Эти условные имена должны быть преобразованы в настоящие имена меток секций. Так же, как и в механизме с переменными, это преобразование происходит на втором этапе, т.е. после генерации секций, в момент объединения сгенерированных секций в единый блок кода.

Теперь нужно уточнить, как работает механизм объединения секций в единый блок кода. Не будем вдаваться в детали синтаксиса, который требуется для того, чтобы блок кода был валидным, лишь уточним как именно различные секции объединяются в единый блок. Напомним, что в самом начале каждой генерируемой секции выдается уникальный id, который используется для генерации уникальных имен переменных и меток внутри секции. После это запускается непосредственно процедура генерации кода секции. После этого секция представляет собой IR-код, который содержит условные имена переменных и меток: %_gX, %_uX, %_cX. В этот момент все секции сгенерированы, и все переменные и метки, которые должны встать на место условных, уже известны. Далее запускается процедура мэппинга условных имен

в настоящие. Для этого каждой секции передаются ассоциативные массивы: globals - содержит пары вида <%_gX, name>, uJumps - <%_uX, name>, cJumps - <%_cX, <name1, name2>>. После замены условных имен на настоящие код секций представляет собой уже полностью корректный IR-код. Далее код секций конкатенируется и оборачивается необходимым синтаксисом, для обеспечения корректности блока кода в целом.

Данная статья содержит лишь часть информации о процессе кодогенерации - в ней подробно описаны механизмы генерации секций, объединения секций в единый блок кода, а также вопросы связанные с выбором необходимых для реализации секций, позволяющих реализовывать определенный набор пользовательских запросов. Однако, за рамками этой статьи остались вопросы эффективной JIT-компиляции, оптимизации и rewriting'а исходного пользовательского запроса для генерации наиболее эффективного кода, вопросы оптимизации сгенерированного IR-кода(по большей части вопросы, связанные с генерацией помимо кода еще и подсказок компилятору), а также вопрос кэширования сгенерированных секций и блоков кода. Эти вопросы будут описаны в будущих статьях.