Промежуточные представления компилятора P4C

Оригинал

Введение

В этом документе рассматривается устройство компилятора P416, описаны разные классы и структуры данных, применяемые для компиляции, а также сам процесс компиляции. Компилятор рассчитан на «традиционную» модель работы, преобразующую программу P4 во внутреннее представление, выполняющую несколько проходов с этим представлением, дающих в результате промежуточное представление (IR[1]), которое более или менее напрямую отображается в код для целевой платформы. По ходу дела набор классов IR, представленных в коде, может меняться, при этом некоторые конструкции будут присутствовать лишь в начальном представлении после синтаксического анализатора, а другие будут добавляться в процессе компиляции.

Новизна заключается в возможности всегда вернуться к более ранней версии IR (обратное прохождение) или использовать множество потоков при работе с IR, что позволяет реализовать разные стратегии при поиске окончательного результата. Поддержка ветвления и возврата является центральным моментом IR и преобразования. Задачей IR является представление программы в виде дерева или DAG[2] с одним корневым узлом и остальными узлами, указывающими лишь своих потомков. Преобразования выполняются путём создания новых узлов IR, являющихся «мелкими» копиями имеющихся узлов, и последующего создания для них новых родительских узлов вплоть до корня IR. Внутреннее представления реализовано на C++ в форме одной иерархии наследования объектов с основным базовым узлом типа Node, используемым всеми субклассами. Этот базовый класс Node поддерживает функциональность, требуемую для преобразований - клонирование, проверку эквивалентности, диспетчеризацию субклассов.

Представление IR должно быть расширяемым, что позволяет добавлять новые классы в иерархию по мере добавления поддержки новых платформ. Исходное представление IR создаваемое синтаксическим анализатором, содержит конструкции, относящиеся лишь к исходному коду P4, и дел него будут выполняться преобразования frontend-компилятором для перевода в более простую и каноническую форму.

IR может быть деревом или DAG, но разрешение ссылок «вперёд-назад» (back/up) в IR (циклах) сводит на нет большинство преимуществ неизменного IR. Если циклы (например, узлы-листья для имён напрямую ссылающиеся на именованный объект) разрешены, практически любое преобразование будет приводить к клонированию всего графа IR, поскольку изменение листа включает клонирование содержащего его объекта, а затем (рекурсивно) любого объекта, указывающего на этот объект.

Проходы Visitor и Transform

Компилятор организован в форме серии проходов Visitor и Transform. Базовые классы Visitor и Transform упрощают определение новых проходов и новому преобразованию требуется лишь указать нужные типы IR, а прочие можно игнорировать. Проход Visitor (постоянный) посещает каждый узел в дереве IR м собирает информацию о дереве, не меняя её, а проход Transform посещает каждый узел и может менять его или заменять иным узлом.

```
/* псевдокод базового преобразования */
visit(node, context=ROOT, visited={}) {
    if (node in visited) {
            if (visited[node] == INVALID)
                    throw loop_detected
            if (VisitDagOnce)
                    return visited[node] }
    visited[node] = INVALID
    copy = shallow_clone(node)
    if (VisitDagOnce)
            forward_children(copy, visited)
    copy = preorder(copy, context) // переопределен в субклассе Transform
    visit_children(copy, {copy, context}, visited)
    copy = postorder(copy, context) // переопределен в субклассе Transform
    if (*copy == *node) {
            visited[node] = node
            return node
    } else {
            visited[node] = copy
            return copy }
}
forward_children(node, visited) {
    for (child in children(node))
            if (child in visited)
                    child = visited[child]
}
visit_children(node, context, visited) {
    for (child in children(node))
            child = visit(child, context, visited)
}
```

В проходе Transform разрешено свободно изменять или полностью заменять узел в своих программах и будет автоматически перестраивать дерево должным образом. Подпрограммы преобразования не могут менять другие узлы в дереве, однако они могут обращаться к ним и ссылаться на них. Эти программы могут быть перезагружены, чтобы

[1]Intermediate representation.

[2]Directed acyclic graph - ориентированный ациклический граф (орграф), в котором отсутствуют направленные циклы, но могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел.

автоматически различать разные субклассы IR. Любому данному посетителю узла нужно лишь переопределить интересующие программы для нужных типов IR.

Имеется несколько субклассов Visitor, описывающих разные типы посетителей.

Субкласс	*Описание*
Inspector	Простой посетитель, собирающий информацию, но не меняющий узлы.
Modifier	Простой посетитель, которые не меняет структуру дерева, но может менять узлы.
Transform	Полное преобразование узла
PassManager	Комбинация нескольких типов, выполняемых последовательно.

Имеется также два интерфейса, которые могут реализовать субклассы Visitor для изменения режима посещения узлов.

Интерфейс	*Описание*
ControlFlowVisitor	Посещение узлов в порядке потока управления, расщепление (клонирование) посетителя по условию и слияние клонов в точке соединения.
Backtrack	Посетитель уведомляется, когда происходит возврат, и может сохранить своё состояние как точку возврата.

ControlFlowVisitor

Для поддержки анализа потока управления имеется специальный интерфейс ControlFlowVisitor, который понимают определённые субклассы IR, инкапсулирующие поток управления, и применяют для управления порядком посещения узлов-потомков и состоянием посетителя, передаваемым его подпрограммам preorder/postorder для анализа потока управления.

Базовая идея заключается в том, что узлы потока управления в IR при посещении своих потомков создают клоны объектов ControlFlowVisitor для посещения различных потомков после ветвления и вызывают функцию flow_merge в точке соединения для слияния состояний посетителя. Базовый класс Visitor и интерфейс ControlFlowVisitor определяют две виртуальные функции - flow_clone и flow_merge решающие эти задачи. В базовом классе Visitor (используется всеми посетителями, *не наследующими* из интерфейса ControlFlowVisitor), где функции реализованы в виде no-ops, функция flow_clone просто возвращает тот же объект, а flow_merge не делает ничего. В интерфейсе ControlFlowVisitor функция flow_clone вызывает функцию clone (которая должна быть реализована классом leaf), а flow_merge является абстрактной виртуальной функцией, которая должна быть реализована в классе leaf.

В качестве примера использования в программах посетителя IR рассмотрим класс IR::If, имеющих 3 дочерних элемента для посещения - утверждение (predicate) и два следствия (consequent), причём значение утверждения определяет, какое из следствий выполняется. Дочерний посетитель для IR::If имеет вид

```
visitor.visit(predicate);
clone = visitor.flow_clone();
visitor.visit(ifTrue);
clone.visit(ifFalse);
visitor.flow_merge(clone);
```

Таким образом оба следствия посещаются с состоянием посетителя, соответствующим потоку управления, сразу после оценки утверждения, а затем состояния потока управления после обоих посещений объединяются.

Для использования этой функциональности субклассу Visitor нужно лишь наследовать от интерфейса ControlFlowVisitor и реализовать функции clone и flow_merge. Другим классам Visitor не нужно делать ничего, но они могут реализовать clone по иным причинам.

Этот анализ потока управления работает лишь с потоком управления «ветвление-слияние» и не подходит для циклов (которые обычно требуют итерации процесса до фиксированной точки). Поскольку IR (в настоящее время) не поддерживает циклы, это не создаёт проблем. Язык P4 не разрешает включать циклы в таблицы «сопоставление-действие» внутри конвейера и это работает достаточно хорошо, но может быть недостаточно для синтаксических анализаторов, которые могут поддерживать циклы.

Поток в целом

Поток компиляции можно грубо разделить на три части - frontend, mid-end и backend. Интерфейсная часть (frontend) выполняет в основном не связанные с целевой платформой преобразования, предназначенные для очистки IR и приведения в каноническую форму. Средняя часть (middleend) выполняет зависящие от целевой платформы преобразования IR. Последняя часть (backend) принимает решения о выделении ресурсов. Поскольку этот этап может завершаться отказом по причине имеющихся ограничений, может потребоваться возврат к mid-end для проверки иной конфигурации. Возврата к начальному этапу (frontend) не предполагается. Разделение на этапы Front/Middle/Back достаточно произвольно и некоторые проходы могут перемещаться между этапами, если в этом будет смысл.

Frontend

Интерфейсная часть (frontend) анализирует исходный код и преобразует его в представление IR, напрямую соответствующее исходному коду. Затем выполняется серия проходов, таких как проверка типов (изменение узлов для указания выведенных типов), раскрытие констант, устранение неиспользуемого кода и т. п. Дерево первоначального IR соответствует исходному коду, а затем выполняются различные (в основном независимые от целевой платформы) преобразования. Сложная структура и сложные области действия в коде здесь преобразуются в «плоское» представление.

Mid-end

На этапе mid-end выполняются преобразования, в той или иной степени зависящие от целевой платформы, но не связанные с выделением конкретных ресурсов. В какой-то части этих преобразований представление IR преобразуется в форму, соответствующую возможностям целевой платформы.

Управление проходами

Для управления возвратами и порядком проходов и преобразования служит объект PassManager, который инкапсулирует последовательность проходов и выполняет их по порядку. Проходы, поддерживающие интерфейс Backtrack, являются точками возврата.

Базовый объект PassManager может управлять динамически создаваемыми последовательностями проходов. Он отслеживает проходы, реализующие Backtrack, и при обнаружении отказа на последующем этапе возвращается к проходу Backtrack.

```
interface Backtrack {
   bool backtrack(trigger);
}
```

Метод backtrack вызывается при возникновении необходимости возврата и возвращает значение true, если в повторном проходе можно проверить применимый вариант. В противном случае (вариантов нет) возвращается false для поиска иной точки возврата.

Использование исключений

Исключения применимы при ошибках, о которых можно сообщить пользователю, проблемах с ограничениями, требующих возврата, и внутренних проблемах компилятора. Для этого создано множество классов исключений.

- Backtrack::trigger служит для управления возвратом и может включать субклассы для добавления информации.
- CompilerBug используется при внутренних ошибках компилятора. Взамен непосредственного вызова служит макрос BUG().

Классы IR

Чтобы упростить управление и расширений для классов IR, они определяются файлах .def, которые обрабатываются программой ir-generator, создающей код IR C++. По сути, файлы .def являются просто определениями классов C++ с удаленным шаблоном, который вставляет программа ir-generator, расщепляя также определение каждого класса на файлы .h и .cpp по мере необходимости.

Основная часть «нормального» IR определяется в каталоге ir, при этом некоторые frontend и backend используют свои расширения имён для IR, определяемых в их подкаталогах внутри ir. Программа ir-generator считывает все файлы .def в дереве и создаёт по одному файлу .h и .cpp, которые содержат весь код определения класса IR.

Все ссылки на объекты класса IR из объектов других классов IR **должны** быть указателями const, чтобы поддерживать инварианты, открытые лишь для записи (write-only). Когда это слишком обременительно, можно встраивать классы IR непосредственно в другие объекты IR, вместо использования ссылок. При таком встраивании объектов IR они становятся доступными для изменения посетителями Modifier и Transform при посещении содержащего объект узла, а сами объекты непосредственно посещаться не будут.

Программа ir-generator понимает оба эти механизма. Любое поле, объявленное в классе IR с другим классом IR в качестве типа, будет преобразовано в указатель const, если у него нет встроенного модификатора (в данном случае это будет встроенный напрямую субобъект).

Программа ir-generator понимает множество «стандартных» методов для классов IR - visit_children, operator==, dbprint, toString, apply. Эти методы можно объявлять *без* аргументов и типов возврата (т. е. просто указывать имя метода с блоком кода в фигурных скобках - {}), при этом будет синтезировано стандартное объявление. Если метод не объявлен в файле .def, создаётся стандартное определение (на основе объявленных в классе полей). Таким способом *большинство* классов могут избежать включения этого шаблонного кода в файл .def.

IR::Node

Это основной абстрактный класс для всех узлов IR, содержащий лишь небольшое количество данных для информирования об ошибках и отладки. В общем случае эти данные **никогда** не сравниваются на предмет равенства (субклассам никогда не следует вызывать Node::operator==`), поскольку узлы (Node), различающиеся лишь этой информацией следует считать эквивалентными и не требовать для них клонирования или дерева IR.

IR::Vector<T>

Этот шаблонный класс содержит вектор указателей (const) на узлы конкретного субкласса IR::Node.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru