Эволюция архитектуры компиляции

Зуев Евгений Александрович,
Swiss Fed Inst of Technology (ETH) Zurich, Switzerland
zueff@inf.ethz.ch

Аннотация

В статье показано развитие систем компиляции в направлении от закрытой, монолитной архитектуры к открытой архитектуре, обеспечивающей поддержку широго спектра операций над прграммами, а также позволяющей интегрировать компилятор в современные среды разработки.

1. Введение. Классическая архитектура компиляции

Традиционная архитектура компиляторов, сложившаяся исторически и описанная в многочисленных источниках, включает совокупность последовательных преобразований исходного текста программы; конечный результат этих преобразований – последовательность команд некоторого процессора («исполнимая программа»), реализующая семантику исходной программы на ЯП. Компилятор в целом рассматривается как монолитная программа, устроенная по принципу «черного ящика»; взаимодействие компилятора с окружением сводится к заданию на входе параметров компиляции и текстов с исходной программой и получению на выходе результатов его работы (рис. 1).

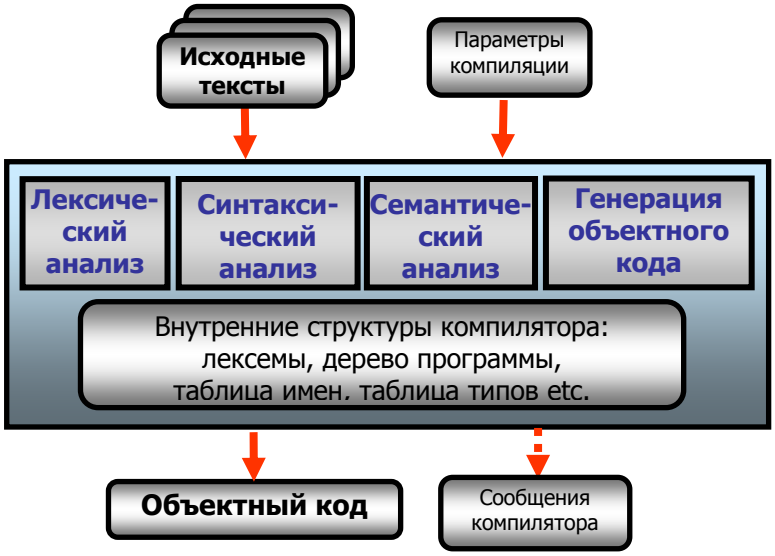

Рис.1 Архитектура традиционного компилятора

Заметим, что структуры данных, формируемые компилятором в процессе работы (дерево программы, таблицы имен и типов и пр.), здесь рассматриваются как сугубо внутренняя информация, необходимость в которой исчезает непосредственно после формирования результата компиляции.

Описанная архитектура, будучи сама по себе вполне логичной и естественной (компилятор в этом смысле практически не отличется от многочисленных «утилит» - служебных программ, выполняющих различные преобразования информации), весьма ограничена в плане основной функциональности и примитивна относительно спектра современных потребностей. В частности, монолитная архитектура существенно ограничивает возможности интеграции компилятора в развитую среду разработки, допуская лишь крайне простые решения вроде обмена информацией посредством дискового пространства. С возрастанием сложности создаваемого ПО и соответствующим ужесточением требований к средам и системам программирования недостатки подобной архитектуры компиляции становятся все более очевидными.

Помимо упомянутой ограниченности монолитной архитектуры, следует указать еще один существенный недостаток классической архитектуры компиляторов. Он заключается в том, что такая архитектура ограничивает как множество входных языков, так и множество целевых платформ. Перенос («перенацеливание») компилятора на новую аппаратную платформу связан с существенными (и часто неприемлемыми) издержками, включающими глубокую переработку значительной части компилятора. В результате такой переработки получается, по существу, новый компилятор, что автоматически удваивает затраты на сопровождение и поддержку. Если же возникает потребность реализовать для данной платформы новый входной язык, то сделать это на основе существующего компилятора другого ЯП практически невозможно – легче разработать новый компилятор с нуля.

Справедливости ради, следует отметить, что описанная проблема была осознана достаточно давно; уже в 70-х годах были предожены продвинутые варианты архитектур компиляции, сводящиеся к разделению языко- и пратформенно-независимых компонент компилятора и введению *промежуточного представления* исходной программы, которое служит интерфейсом между компонентами (рис. 2).

Рис. 2 Многоплатформенная архитектура компиляции

Возможно, наиболее известной многоязыковой переносимой архитектурой является система gcc (GNU Compiler Collection [1]), разработанная под эгидой консорциума FSF.

Можно утверждать, что подобная архитектура (которая к настоящему времени уже стала, в свою очередь, классической) вполне практична для целей создания многоязыковых систем программирования для большого спектра аппаратных платформ. Успех gcc позволяет сделать вывод о существенном сокращении затрат в рамках этой архитектуры: добавление компилятора переднего плана для некоторого нового языка автоматически приводит к появлению компиляторов с этого языка для всех аппаратных платформ, поддерживамых архитектурой. Аналогично, включение в систему генератора кода для новой платформы, по существу, означает добавление целого семейства компиляторов со всех имеющихся в системе входных языков для этой новой платформы.

Описанная архитектура получила широкое распространение, а концепция выделения в компиляторе языко-ориентированных и платформенно-ориентированных частей стала повсеместной. Помимо упомянутых достоинств, явное выделение компонент компилятора в виде отдельных модулей приводит к большей гибкости архитектуры и,

как следствие, частично решает задачу интеграции компилятора в окружения разработки.

Один из результатов утверждения такой архитектуры состоит в том, что особое внимание исследователей и разработчиков было обращено на ее центральный элемент – *промежуточное представление* (ПП). За последние годы было предложено большое число различных подходов к построению ПП – от низкоуровневых списковых структур, описывающих обобщенные команды типовых процессоров, до реализации ПП в виде реляционной базы данных. Заметим, что появившиеся в последнее время архитектуры Java byte code и Microsoft Intermediate Language (MSIL) платформы .NET также (с некоторыми оговорками) могут быть также отнесены к ПП.

Однако, несмотря на то, что концепция ПП принята практически повсеместно, как соответствующая современному пониманию целей и задач компиляции ЯП, значительное большинство реализаций этой концепции страдает двумя взаимосвязанными недостатками, которые образуют *фундаментальный изъян* современных систем программирования. Эти недостатки следующие:

- Низкий уровень и бедная структура большинства ПП, отсутствие в них существенной информации об исходной программе (прежде всего, семантической информации) и, как следствие

- Поддержка, по существу, единственной задачи – генерации объектного кода.

По нашему мнению, эти недостатки носят принципиальный характер, во-первых, потому, что они делают большинство известных ПП неадекватными современным требованиям, предъявляемым к системам программирования, и, во-вторых, потому, что они не могут быть преодолены в рамках существующих подходов.

2. Современный взгляд на задачу компиляции: семантическое представление программы

Легко видеть, что архитектуры компиляции, описанные выше, решают, по существу, единственную задачу – *получение машинного кода*, пригодного для исполнения на некотором процессоре. Расширенная архитектура в этом смысле отличается от первоначальной только числом целевых процессоров.

Соответственно этой задаче (далее будем называть ее компиляцией в собственном, или узком смысле) создается и промежуточное представление. В огромном большинстве случаев оно представляет собой варианты «обобщенных ассемблеров» низкого уровня, на основе которых достаточно легко генерировать реальные машинные коды для

конкретных процессоров. Из-за такой крайне узкой специализации ПП их чрезвычайно неудобно (а часто и невозможно) использовать для каких-либо иных целей, кроме генерации кода. Традиционная структура ПП затрудняет реализацию даже такой крайне необходимой на практике задачи, как поддержка отладки программ в терминах исходного ЯП – для этого требуются специальные усилия по расширению ПП (добавление так называемой «отладочной информации»).

В то же время современные проблемы разработки программных систем предполагают существенно более широкую трактовку задачи компиляции – не только как генерацию исполняемого кода, но и как *широкий спектр различных операций над текстами программ*. Не вдаваясь в детальное обсуждение (см., например, [2]), можно кратко обозначить следующие типичные задачи, связанные с обработкой программных текстов:

- *Компиляция* в узком смысле (генерация исполнимого кода);

- задачи, связанные с *пониманием* программ человеком (несколько упрощая, эти задачи можно назвать **визуализацией**);

- *верификация* программ;

- *статический анализ* программ;

- *интерпретация* программ.

Можно уверенно говорить, что перечисленные задачи принадлежат к числу важнейших современных потребностей индустрии ПО. Существенное обстоятельство, которое роднит все эти задачи, заключается в том, что их решение возможно на основе **семантического анализа программ**. В то же время, легко видеть, что именно полный семантический анализ программы (совместно с лексическим и синтаксическим анализом) и составляет существо традиционного понимания компиляции, а результат такого анализа, как правило, представляется в виде совокупности структур данных (дерева программы, таблиц имен и др.), формируемых типичным *компилятором переднего плана* в процессе обработки исходной программы.

Важнейший вывод из проведенных рассуждений заключается в том, что содержание внутренних структур данных, формируемых в процессе компиляции, может (и должно) служить информационной основой для решения задач, перечисленных выше. Промежуточное представление программы, основаное на семантических структурах компиляции, правильнее было бы назвать *семантическим представлением* (СП) программы. В отличие от типичного промежуточного представления, СП несет в себе полное знание об исходной программе, в том числе, и такие его аспекты, которые присутствуют в исходной программе неявно. С другой стороны. оно содержит только ту информацию,

которая относится к семантике исходной программы, и не включает каких-либо узкоспециализированных структур (например, объектный код). Тем самым, СП может выступать в роли универсального интерфейса между компилятором переднего плана и разнообразными языко-ориентированными компонентами систем разработки (см. рис. 3).

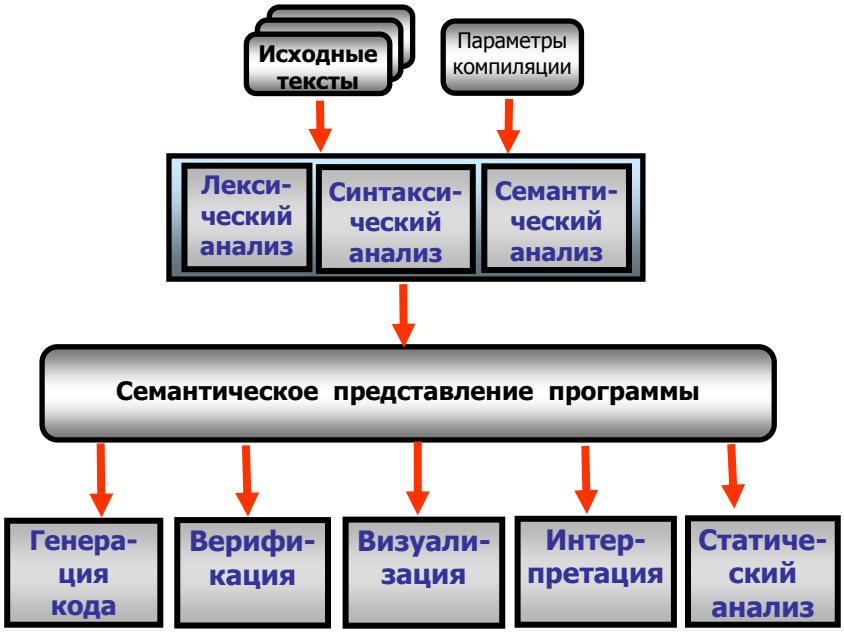

Рис. 3 Архитектура, основанная на семантическом представлении

В простейшем случае СП может представлять собой традиционные структуры компиляции, некоторым образом сохраненные в долговременной памяти. Однако подобный «прямой» путь формирования СП далеко не всегда приемлем, прежде всего из-за чрезмерной сложности интерфейса со структурами компиляции. В следующем разделе этот вопрос обсуждается более подробно.

3. XML как технологическая основа семантического представления

В данном разделе обсуждаются в основном практические аспекты, связанные с наиболее адекватным форматом семантического представления.

Возвращаясь к понятию промежуточного представления программы как предшественника концепции СП, введенной в предыдущем разделе, следует отметить, что в огромном большинстве конкретных случаев

формат ПП является двоичным и, следовательно, закрытым. Это означает, что единственная возможность получить доступ к такому ПП – воспользоваться специально разработанным для него программным интерфейсом (API).

Вообще говоря, программный интерфейс – принципиально универсальный способ доступа, с помощью которого можно реализовать любую требуемую функциональность (то есть, любой алгоритм анализа семантики исходной программы), и во многих случаях его использование является оптимальным решением. Однако в перспективе, по мере появления новых видов потребностей, программного доступа к СП может оказаться недостаточно. Реализация нового языкового процессора на основе API предполагает разработку полноценной (и, как правило, нетривиальной) программы на языке, из которого возможен доступ к «семантическому» API, что в ряде случаев может оказаться для клиента невозможным или неприемлемым – либо в силу недостаточной квалификации, либо с точки зрения временных затрат, либо по другим причинам. Кроме того, в принципе невозможно предугадать спектр потребностей клиентов, желающих использовать СП для извлечения необходимой им информации о программе; вполне возможно, что для этого удобнее использовать иные, более простые и прямые пути, нежели разработка программы, основанной на API.

Коротко, существо решения, снимающего потенциальные проблемы с доступом к СП, можно сформулировать следующим образом: *сделать формат СП открытым*.

Такое решение дает ряд существенных преимуществ. Во-первых, открытый формат предполагает наличие стандартных и общедоступных средств доступа. Во-вторых, эти средства доступа (в случае достаточно распространенных и популярных форматов) имеются для различных платформ, что автоматически делает решения, основанные на них, переносимыми. В-третьих, для открытого формата, как правило, имеется целый спектр средств доступа различного характера: от обычного API до высокоуровневых специализированных пакетов. Кроме того, для открытого формата можно создать собственные интерфейсы, если имеющихся возможностей недостаточно для конкретных целей.

Наконец, немаловажное обстоятельство заключается в том, что все перечисленные возможности поддерживаются независимыми организациями (зачастую, эти форматы представляют собой просто международные или отраслевые стандарты), что делает решения, основанные на них, жизнеспособными в долговременной перспективе.

Возвращаясь к семантическому представлению, можно утверждать, что наиболее привлекательным открытым форматом для ПП в настоящее

время является формат XML [3]. Все перечисленные выше достоинства открытых форматов в полной мере применимы к XML. В самом деле, в настоящее время XML представляет собой промышленный стандарт де-факто, признаваемый и поддерживаемый всеми без исключения ведущими мировыми производителями ПО, а также сотнями небольших компаний. Для доступа к XML-документам имеется целый спектр средств доступа (большинство из которых, в свою очередь, стандартизовано) – от традиционных API (на основе спецификаций DOM и SAX) до мощных специализированных технологий преобразования (XSLT) и языков запросов (XQuery). Конфигурация средств доступа к СП, представленному в формате XML, иллюстрируется рис. 4.

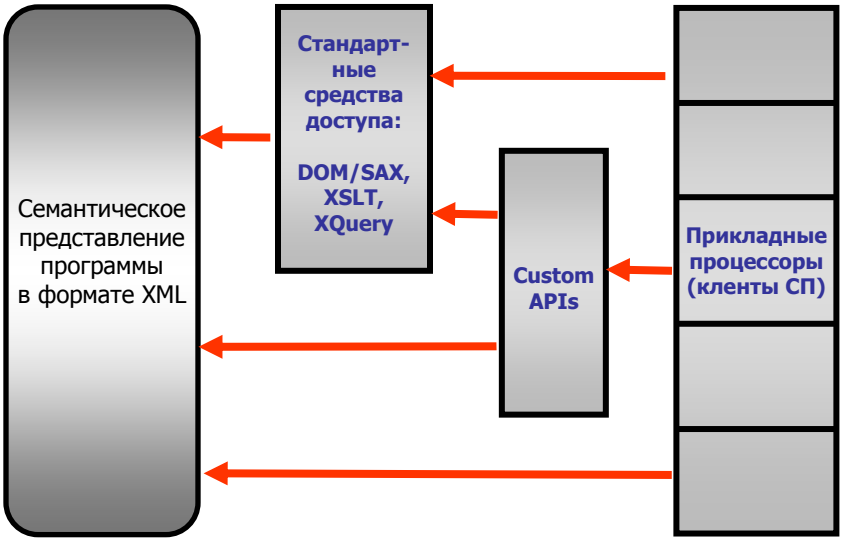

Рис. 4 Конфигурация способов доступа
к семантическому представлению

Наконец, принципиально важным является то обстоятельство, что язык XML как таковой, в силу своей иерархической структуры, универсальности и расширяемости, удивительно хорошо подходит для представления всех видов информации, относящейся к ПО,- от синтаксической структуры до разнообразных семантических связей между элементами программы. Более того, способность XML и технологий, основанных на нем, представлять и эффективно обрабатывать слабоструктурированную и неструктурированную информацию (в случае использования для СП это комментарии, спецификации, выраженные в виде неформальных текстов, информация о версиях и вариантах, задания на компиляцию и сборку и многое

другое) делает этот формат очень мощным и потенциально единственным средством представления не только программ как таковых, но и всей информации, относящейся к программному проекту.

4. Интегрируемая архитектура компиляции

В данном разделе мы кратко (насколько позволяет объем статьи) коснемся еще одного аспекта, оказывающего заметное влияние на современные архитектуры компиляции. Речь идет о задаче интеграции компилятора в среды разработки.

Современное понимание указанной задачи далеко не исчерпывается обеспечением совместного функционирования компилятора, редактора связей и отладчика в рамках единой системы. Как правило, бывает необходимо полноценное «понимание» системой лексической и синтаксической структуры программы (в частности, наглядное выделение лексических элементов программы, механизм автозавершения конструкций, оперативное выявление синтаксических ошибок), динамический анализ семантических связей внутри программы в процессе ее редактирования (например, контекстно-чувствительная идентификация типов, включая составные типы, с поддержкой оперативного доступа к их элементам).

Реализация возможностей, подобных описанным выше, становится возможной при условии существенных изменений в архитектуре компиляции. Существо этих изменений заключается в том, что традиционные этапы компиляции - лексический, синтаксический и семантический анализ - трактуются не как внутренние фазы его функционирования, а рассматриваются как отдельные компоненты, допускающие *независимую активизацию* как со стороны других компонент, так и со стороны интегрирующей платформы. Иными словами, среда разработки может в любой момент своей работы (сообразуясь с собственной логикой функционирования) вызвать, например лексический анализатор, передав ему в качестве исходной информации некоторый фрагмент исходной программы, получить информацию о лексемах, содержащихся в пределах этого фрагмента, а затем передать эту информацию текстовому редактору, который отобразит на экране обновленный фрагмент исходного текста в наглядном виде.

Аналогично, система может организовать, параллельно с вводом программы разработчиком, «фоновую» компиляцию вводимых фрагментов текста с вызовом лексического и синтаксического анализаторов и оперативным «расцвечиванием» синтаксической структуры и выводом, если необходимо, диагностических сообщений.

Достаточно ясно, что наиболее адекватной основой для подобной архитектуры служит объектно-ориентированный подход. В этом случае упомянутые компоненты компилятора представляются в виде классов, реализующих некоторые стандартные интерфейсы, которые определены в среде разработки. Последняя может произвольным образом (согласно собственной логике) создавать экземпляры этих компонент и активизировать их функциональность согласно определенным интерфейсам. Две наиболее мощные и известные среды - система разработки Microsoft Visual Studio .NET и универсальная интегрирующая платформа Eclipse [4] - поддерживают именно такой стиль интеграции новых компонент.

5. Заключение

Современный взгляд на архитектуру систем компиляции в значительной степени определяется двумя группам потребностей: во-первых, необходимостью поддержки широкого спектра операций над программами (в котором такая традиционная операция, как генерация исполнимого кода, является лишь одной из многих) и, во-вторых, необходимостью глубокой интеграции компиляторов с другими языко-ориентированными процессорами и с развитыми средами разработки.

В статье показано, что решение первой задачи обеспечивается выделением в компиляторе языко-зависимых компонент (образующих компилятор переднего плана) и введением понятия семантического представления, которое в удобном для доступа виде содержало бы полное знание о семантике исходной программы. Обосновывается выбор открытого формата для семантического представления, наиболее подходящим вариантом которого является формат XML.

Интеграция компилятора в современные среды разработки также диктует необходимость изменения традиционной архитектуры компиляции. Эти изменения заключаются в последовательном применении принципов ООП, согласно которым компоненты компиляторов трактуются как независимые классы, реализующие совокупность стандартных интерфейсов.

Литература

1. GCC Home Page, http://gcc.gnu.org.

2. Зуев Е.А. Принципы и методы создания компилятора переднего плана Стандарта Си++. Диссертация на соискание ученой степени кандидата физико-математических наук, Москва, 1999.

3. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04Feb 2004, www.w3c.org/TR/2004/REC-xml-20040204/.

4. Eclipse Home Page, www.eclipse.org.