Semantic APIs for Programming Languages

Eugene Zouev
Interstron, Moscow
eugene.zueff@gmail.com

Abstract (English)

The evolution of approaches to the design of advanced language oriented tools for non-compilation manipulations on programs is discussed. The notion of semantic representation (SR) as the basis of such tools is introduced. SR is considered as a composition of information structures together with corresponding functionality which holds the comprehensive knowledge about the program semantics. The major components of the semantic representation of C++ programs (project SemantiC++) are considered and discussed. The most important and novel features of the SemantiC++ representation are the support for SR generation incorporated into the SR and the semantic search feature which enables powerful and sophisticated semantic queries to the SR in an easy and elegant way.

***Keywords**: Programming Languages' Semantics; Semantic Representation;API;C++.*

Семантические интерфейсы языков программирования

Евгений Зуев
Компания Интерстрон
Москва
eugene.zueff@gmail.com

Abstract (Russian)

В статье проанализирована эволюция подходов к разработке продвинутых языковых инструментов для выполнения некомпилирующих операций над программами, потребность в которых в настоящее время является крайне актуальной. Как основа подобных операций и соответствующих программных инструментов вводится понятие семантического представления (СП) программ как композиции информационных структур и соответствующей функциональности, ориентированного на всеобъемлющее представление знаний о семантических свойствах программ. Рассматриваются важнейшие составляющие программного интерфейса СП для Си++, создаваемого в рамках проекта SemantiC++. Обосновывается включение в интерфейс СП функциональности генерации и семантического поиска, что составляет новизну предлагаемого подхода.

***Keywords**: Семантика языков программирования; семантическое представление; программные интерфейсы; Си++.*

1. Введение. Язык Си++ и его инструментарий

В последние годы в индустрии ПО появилось большое число языков программирования, некоторые из которых получили значительную популярность. Однако, несмотря на активное развитие таких средств программирования, как Java, C#, Visual Basic и многих других, несмотря на появление других, весьма перспективных и привлекательных инструментов,- язык Си++ остается одним из наиболее распространенных и активно используемых. Согласно индексу TIOBE на июль 2010 года [1], язык Си++ (вместе со своим предшественником Си) уступает в распространенности только языку Java, причем такое положение остается неизменным на протяжении последних десяти лет.

Распространенность Си++ как средства разработки программ находится, на наш взгляд, в серьезном противоречии с ситуацией в области инструментария для этого языка, предлагаемого индустрией. Несмотря на серьезную эволюцию в подходах к процессу создания, тестирования и

сопровождения программ, соответствующий инструментарий (компиляторы, редакторы связей, отладчики и т.д.) в большинстве случаев представляет собой набор примитивных по архитектуре утилит, предоставляющих явно недостаточный на настоящий момент набор возможностей.

Архаичная архитектура формирования многомодульных программных конфигураций (механизмы импорта/экспорта программных сущностей, основанные на текстовых макросах), характерная для языка Си и в полной мере унаследованная Си++, существенно затрудняет анализ программ, снижает качество диагностики ошибок, препятствует эффективному использованию новых языковых возможностей (в частности, шаблонов) и зачастую делает невозможным создание продвинутых языковых инструментов, насущно необходимых индустрией.

Современные интегрированные среды разработки, за очень немногими исключениями, не обеспечивают качественного изменения ситуации, оставаясь до сих пор простым соединением тех же традиционных утилит с визуальными редакторами.

По нашему мнению, существо проблемы заключается в архаичном, но глубоко укоренившимся в индустрии подходе к программам как текстовым объектам. Такой подход, исторические причины возникновения и становления которого вполне очевидны, составляет наиболее серьезное препятствие для качественного улучшения дел с поддержкой процесса разработки программ.

Отмеченное выше противоречие между возможностями языка как такового и ограниченностью соответствующего инструментария характерно практически для любого языка программирования. Но, как уже говорилось, наиболее ярко это противоречие проявляется для Си++, прежде всего, из-за особенностей его появления и развития. Хорошо известно, что одним из важнейших требований при проектировании Си++ заключалось в обеспечении совместимости с его предшественником – языком Си. Заметим, что требование совместимости проистекало прежде всего из необходимости использования существующих на тот момент инструментов, в частности, компиляторов и редакторов связей. Таким образом, проблемы, связанные с инструментарием языка Си++, были заложены изначально при его проектировании. К настоящему времени они превратились в фактор, препятствующий созданию действительно мощных и современных инструментов для Си++ и тем самым серьезно снижающий эффективность использования этого языка.

В своей книге [2] Б.Страуструп таким образом характеризует фундаментальное противоречие между языком и его инструментарием:

«…Люди в большинстве своем смотрят на С++-программу как на набор исходных файлов или строк символов. **Программа - это набор типов, функций, предложений и т.д.** Данные понятия представляются в виде символов в файлах только для удобства изображения в традиционных средах программирования.»

Подход к программам на Си++, предлагаемый Б.Страуструпом в этом высказывании, естественно назвать **семантическим**, в противоположность традиционному текстовому подходу. Далее в данной статье делается попытка показать существенные выгоды последовательного и систематического применения семантического подхода к программам как таковым и к типичным операциям над ними.

Для удобства последующих рассмотрений ниже представлена типичная структура традиционного языкового инструмента (несущественные для данного изложения детали опущены):

Рис. 1 Монолитный компилятор

Архитектура монолитного компилятора представляет собой классический «черный ящик»: это замкнутый языковой процессор, который ориентирован на выполнение конкретной задачи порождения кода, предназначенного для выполнения на конкретной аппаратной архитектуре, для заданной входной программы. Таблицы трансляции, несущие семантический образ исходной программы, представляют собой внутреннюю информационную структуру компилятора, предназначенную исключительно для выполнения семантических проверок и собственно генерации кода. В ходе компиляции и генерации таблицы могут пополняться и сокращаться согласно внутренней логике алгоритмов компилятора; целостного образа исходной программы ни в один момент, вообще говоря, не существует.

2. Семантическое представление программ

Прежде чем перейти к обсуждению семантического представления программ, сформулируем базовый тезис данной статьи, на основе которого будут проводиться последующие рассмотрения.

Развитие научных основ компиляции и накопленный практический опыт разработки компиляторов привели к тому, что задача генерации высококачественного кода, пригодного для исполнения на процессоре, решается, как правило, на достаточно высоком уровне. В то же время, возрастающие потребности индустрии разработки ПО приводят к необходимости создания различных языковых инструментов некомпилирующего назначения, адекватных этим потребностям.

Генерация кода из единственной задачи компиляции превращается только в одну (и зачастую не главную) задачу из широкого спектра разнообразных семантических операций над текстами программ на ЯП.

Попросту говоря, мы научились достаточно быстро и с приемлемым качеством порождать объектный код для программ, написанных на языках высокого уровня (то есть, решать задачу компиляции в узком смысле). Однако, в том, что касается *понимания* (в широком смысле) создаваемых и используемых программ, прогресс явно недостаточен.

Подробное обоснование этого тезиса, а также обзор характерных и наиболее актуальных задач компиляции в широком смысле (статический анализ, визуализация, верификация, прямой и обратный инжиниринг, интерпретация и т.д.) содержится в работе [3, Введение]. Там же в качестве базиса для решения перечисленных задач предложена концепция **семантического представления** – информационной структуры, содержащей всеобъемлющую информацию о программе, которую можно извлечь из ее обычного (текстового) представления – включая и скрытую семантику, присутствующую в программах неявно, но наличие и характер которой задается каноническим описанием языка программирования.

Семантическое представление программы представляет собой сложно организованную информационную структуру (в общем случае, граф), элементы которой (узлы графа) представляют все виды сущностей, образующих понятийный базис используемого языка программирования, а связи между элементами (ребра графа) отражают возможные виды отношений между сущностями. Для случая Си++ сущностями служат такие понятия, как переменные, типы/классы, функции, шаблоны, выражения, операции и операторы. Отношения включают структурную подчиненность (для областей действия), принадлежность (для членов классов и локальных объектов), наследование (для классов), совместное использование (для функций и операций), перекрытие (для виртуальных функций), настройки и специализации (для шаблонов), связи вида «объявление-использование» и другие. Примерами скрытой семантики программ на Си++, которая должна получить свое отражение в семантическом представлении, могут служить порядок и особенности вызова конструкторов при инициализации массивов и классовых объектов, правила разрушения локальных объектов при выходе потока управления из области действия или при свертке стека (stack unwinding).

В целом семантическое представление должно содержать полное знание о программе, которое может быть извлечено в процессе анализа ее текстового представления.

3. Семантика программ и таблицы компиляции

Семантическое представление имеет очевидные аналоги с внутренними таблицами традиционного компилятора. В самом деле, для успешной и эффективной генерации кода компилятор должен выявить всю необходимую информацию о смысле входной программы. Эта информация сосредотачивается в его внутренних структурах, важнейшими из которых служат абстрактное синтаксическое дерево и таблицы символов. В принципе, эти структуры можно рассматривать как вариант семантического представления. Как говорит Б.Страуструп в цитировавшейся выше книге,

«В программе на C++ есть много информации, которая в типичной среде доступна только компилятору. **Уверен, что она должна быть предоставлена программисту**.»

Высказанная идея представляется совершенно правильной, однако имеется ряд обстоятельств, затрудняющих ее реализацию или делающих ее вообще невозможной. Во-первых, информация, собираемая компилятором в процессе анализа исходной программы, в типичном случае предназначена исключительно для нужд самого компилятора (в первую очередь, для контроля ошибок и для генерации кода) и потому может быть семантически неполна и не слишком удобна

для любых нестандартных манипуляций. Во-вторых, зачастую внутренние информационные структуры компилятора просто невозможно использовать, так как они практически никогда не существуют как единое связанное целое. Например, внутреннее представление в компиляторе Си++ из пакета Visual Studio создается в процессе анализа программы по частям (например, для отдельных функций) и по частям же удаляется (после генерации кода для функции). Выполнить какие-либо содержательные действия, связанные с анализом программы в целом, пользуясь таким фрагментарным представлением, невозможно. В-третьих, даже если разработчики компилятора предоставляют доступ к промежуточному представлению (например, Ада-компилятор GNAT [5] может сохранить на дисковом файле дерево программы целиком), они принципиально не дают никаких гарантий, что в очередной версии компилятора формат этого дерева не изменится.

Несмотря на указанные проблемы, подход, обеспечивающий программный доступ к семантической информации из внутренних структур компилятора, в ряде случаев образует эффективную архитектуру, давая возможность использовать собранную компилятором информацию для построения ряда практически полезных инструментов.

Такая архитектура предусматривает выделение языко-независимой части компилятора (его «передний план») в виде отдельного компонента. Результат его работы - промежуточное представление программы, хранящее с той или иной степенью полноты ее образ,- служит интерфейсом между передним планом и конечными компонентами, например, генераторами кода для различных аппаратных платформ. Само промежуточное представление (ПП) вместе со средствами доступа к нему выступает в качестве отдельного логического компонента системы.

В зависимости от целевой ориентации многокомпонентной системы компиляции, ПП может быть как сравнительно низкоуровневым и в таком виде пригодным к ограниченному использованию (например, генерация кода для нескольких целевых процессоров, отладка и т.п.), так и более полным с точки зрения отражения семантики исходной программы. В последнем случае спектр возможных конечных процессоров, использующих такое ПП, может включать и более продвинутые средства, например, статического анализа.

Примером использования в этой схеме «полного» ПП может служить компилятор переднего плана компании Интерстрон [6, 7], в котором семантические таблицы вынесены из компилятора переднего плана в отдельную компоненту, снабженную собственным программным интерфейсом. Построенное промежуточное представление (ПП), несущее полную информацию о семантике программы на Си++, может использоваться различными практически полезными языковыми инструментами, среди которых статические анализаторы, верификаторы, генераторы тестовых покрытий, визуализаторы, а также инструменты для исполнения программ в специальном (например, в отладочном) режиме (виртуальная машина Си++).

Принципиальная схема такой архитектуры компиляции показана на следующем рисунке.

Рис. 2 Промежуточное представление как отдельный компонент

Представленная архитектура представляет собой несомненный шаг вперед, однако ей присущ ряд недостатков, которые не позволяют считать ее полностью приемлемым решением. Необходимо отметить следующие особенности:

Во-первых, промежуточное представление программы, пусть даже оно реализовано в виде отдельного компонента и снабжено программным интерфейсом (API), все равно остается «таблицами компилятора», пусть даже несколько облагороженными за счет удачно спроектированного API. Исторически их основное назначение – все-таки компиляция в узком смысле; их структура «подогнана» прежде всего под потребности генерации и/или оптимизации объектного кода.

Во-вторых, даже если конкретная конфигурация «компилятор переднего плана – промежуточное представление» обеспечивает полное отображение семантики входного языка, программный интерфейс к этому представлению может быть весьма ограниченным по своим возможностям. В качестве примера можно привести семантический интерфейс ASIS [4] для языка Ада. Будучи разработанным специально для полного отображения семантики языка Ада, этот интерфейс допускает доступ исключительно по чтению. Иными словами, любая реализация стандарта ASIS (например, в связке с компилятором GNAT [5]) не обеспечивает возможность программной модификации информационных структур, «спрятанных» за

интерфейсом. Модифицировать промежуточное представление Ада-программы можно только путем соответствующей модификации исходного текста и повторного запуска компилятора. Это неизбежно усложняет любую технологию, основанную на ASIS, и существенно ограничивает спектр инструментов, которые можно построить с помощью этого интерфейса.

В-третьих, данная архитектура безусловно предполагает наличие компилятора переднего плана – именно он генерирует промежуточное представление. Поэтому она диктует однонаправленный порядок использования по схеме «исходный текст -> ПП -> конечная информация». Любые менее тривиальные схемы, предполагающие циклическую разработку (например, инкрементальная компиляция) приводят к неоправданному усложнению технологии: так, при внесении изменений в ПП (даже если API данного конкретного ПП допускает его модификацию) приходится синтезировать по нему текст и отдавать его на компиляцию стороннему компилятору, который сгенерирует новое ПП.

4. От промежуточного представления к семантическому представлению

Отмеченные недостатки приводят к необходимости разработки семантического представления, не связанного с концептуально аналогичными информационными структурами, свойственными компиляторам, а также обладающими собственной функциональностью, необходимой для поддержки типичных операций над этим семантическим представлением. Источником такого «настоящего» семантического представления должна служить семантика входного языка. Согласно Б.Страуструпу, «единственная фундаментальная константа - это базовая семантика языка.»

Следует отметить, что понимание необходимости описанного выше подхода к разработке семантического представления возникло не вчера. Так, проект SAGE, ориентированный на представление семантики Си++, начался еще в середине 90-х годов. С тех пор он проделал значительную эволюцию, результатом которой в настоящее время является система ROSE [8]. Близкий подход используется в системе Pivot [9], развиваемой при участии Б.Страуструпа. Общность подходов, характерная для этих проектов, заключается в абстрагировании от традиционных «таблиц компиляции»; дизайн семантического представления отталкивается прежде всего от семантики самого языка и потребностей прикладных инструментов.

Однако, эти проекты, к сожалению, характеризуются одним фундаментальным недостатком, унаследованным от систем предыдущего поколения. Этот недостаток заключается в отсутствии собственного механизма генерации СП. Для этих целей используется сторонний компилятор переднего плана компании Edison Design Group [10], выходные структуры которого конвертируются в «родное» семантическое представление. В результате технологичность связок EDG-ROSE и EDG-Pivot существенно снижается: любая модификация СП влечет необходимость регенерации для измененного СП текста программы с целью ее повторной компиляции с помощью EDG и последующей конвертации результата в СП.

Таким образом, общий недостаток и слабость упомянутых подходов заключается прежде всего в их привязке к компилятору, который выступает в этой связке как сторонняя независимая программа.

Тем самым, мы приходим к осознанию факта безусловной первичности собственно СП перед какими бы то ни было программными агентами, его порождающими: по сравнению с важностью проектирования полного, удобного и адекватного представления семантики входного языка не столь существенно, как именно создается это представление. Прообразом семантического представления может служить как исходный текст программы или ее части, так и, например, UML-диаграмма класса, построенная с помощью некоторого независимого инструмента. Вполне допустимой операцией может оказаться, скажем, построение СП программным путем «с нуля» и его объединение («линковка») с некоторым существующим СП. Таким образом, концептуально функциональность генерации представления рассматривается просто как часть общей функциональности интерфейса СП наряду с другими необходимыми действиями, например, валидацией.

Эти соображения приводят к следующему фундаментальному решению: разработать дизайн семантического представления, после чего спроектировать его программный интерфейс, причем действия по генерации семантического представления (традиционно ассоциируемые с компилятором переднего плана) представить не в виде отдельной процедуры, а как *часть функциональности семантического интерфейса*. При таком подходе функциональность генерации должна играть подчиненную, вторичную роль; в качестве же основного компонента должен

выступать именно семантический интерфейс как таковой. Такой подход к структурной организации языковых процессоров может быть проиллюстрирован следующим рисунком:

Рис. 3 Семантическое представление как самодостаточная сущность

Семантическое представление программ на Си++, представляемое в последующих разделах, спроектировано и реализовано согласно принципиальной схеме, приведенной на рисунке выше. Это СП разработано в рамках проекта SemantiC++, работа над которым ведется в последние несколько лет. Далее кратко описывается устройство семантического представления, а также его базовая функциональность.

5. Семантическое представление программ в проекте SemantiC++

Программно СП проекта SemantiC++ представляет собой множество классов, которые служат представителями понятий входного языка. Каждый класс СП соответствует одному понятию языка; при этом множество таких понятий включает не только программные конструкции Си++ (например, объявления, операторы, выражения, типы), но и такие сугубо семантические понятия, как «контекст», «область действия», «доступность». Отношения наследования и агрегации классов СП строго соответствуют структурным и семантическим отношениям понятий входного языка. Тем самым, для использования СП достаточно знать входной язык; СП как таковое не привносит никаких собственных понятий.

6. Базовая функциональность интерфейса SemantiC++

Богатство изобразительных возможностей СП не исчерпывается отображением иерархических отношений между программными сущностями; классы-представители сущностей не являются «чистыми» структурами данных (что характерно, например, для близкой по задачам системы Common Compiler Infrastructure (CCI, [11, 12]). Напротив, классы СП SemantiC++ обладают развитой функциональностью, важнейшими составляющими которой служат:

- Генерация СП;
- Валидация;
- Семантический поиск;
- Интерпретация;
- Механизмы расширения СП

Остановимся на первых трех составляющих более подробно.

6.1. Стратегии генерации СП

СП в проекте SemantiC++ образуется в результате применения статических функций `open()` или `create()` для класса, представляющего открываемую или создаваемую сущность языка. Указанные функции реализуют две равноправные стратегии генерации СП.

Первая функция служит для создания фрагмента СП на основе некоторого текста или другой входной информации, представленной в виде абстрактного входного потока. Можно сказать, что совокупность статических функций `open()` и образует синтаксический анализатор (парсер) в его традиционном понимании. При этом, что принципиально важно, каждый такой метод реализует только «собственную» часть функциональности парсинга, относящуюся к сущности Си++, представляемой классом-владельцем open(). Так например, функция open() класса `DECLARATION`, отображающего общее понятие объявления языка Си++, реализует разбор конструкции, описываемой синтаксисом объявлений, а такая же функция из класса `PROGRAM` производит разбор всей программы согласно грамматике для головного правила *translation-unit*. В обоих примерах исходный текст, подлежащий разбору, представляется классом `SOURCE`, абстрагирующим понятие входного потока. Этот класс считается частью интерфейса СП.

Необходимо сделать существенное уточнение. Если для «традиционного» случая, когда на вход анализатору подается программа как единое целое, достаточной информаций (отвлекаясь от технических деталей вроде опций и параметров разбора) является упомянутый «входной поток», то для случая разбора, например, одиночного объявления или последовательности операторов языка этого недостаточно. А именно, необходима информация о **контексте** разбора, относительно которого должны разбираться подаваемые на вход конструкции языка и разрешаться вхождения имен. Такой контекст, который можно упрощенно описать как область действия, содержащую обрабатываемую конструкцию, вместе с совокупностью объемлющих областей действия, представляется специальным классом `CONTEXT`. Этот класс, представляя *область*

действия (scope)- одну из фундаментальных концепций Си++ - также входит в состав интерфейса СП.

Открывать по отдельности можно такие конструкции, как объявления/описания переменных, объявления/описания функций и функций-членов, объявления классов и перечислимых типов, шаблоны функций и классов и даже выражения и отдельные операторы. Поэтому функции `open()` определены только для классов СП, представляющих перечисленные конструкции входного языка. Контекстом разбора в перечисленных случаях служит ссылка на область действия, объемлющую обрабатываемую входную конструкцию. Тем самым, понятие частичной компиляции из весьма сложной в традиционной схеме операции становится лишь частным случаем формирования СП.

В отличие от `open()`, статические функции-члены `create()` служат для создания фрагментов СП без какого-либо прототипа - «с нуля». Это означает, что можно программно сформировать СП, которое будет представлять семантический «образ» некоторой (реально не существующей) программы, задав все ее структурные части, а также семантические свойства и отношения. В отличие от `open()`, функции `create()` определены для всех классов СП.

Разумеется, всегда можно комбинировать два способа генерации СП: открытие и создание. В частности, можно открыть СП некоторой (части) исходной программы, после чего дополнить его созданными «вручную» фрагментами. Вообще говоря, допустимы любые комбинации СП, имеющего некоторый прототип в виде программного текста, с фрагментами СП, созданными «с нуля». Такой смешанный механизм формирования семантического представления, основанный на свободной суперпозиции программных вызовов `open()` и `create()`, очень гибок и предоставляет большой спектр возможностей для реализации различных стратегий работы с СП. При этом от программиста не требуется каких-либо специальных знаний: необходимо только хорошее понимание структурного устройства программ на Си++, а также известная аккуратность в манипулировании семантическими представлениями различных конструкций.

Завершая краткое рассмотрение функций генерации, можно сделать вывод, что в СП изначально заложены механизмы, аналогичные известному подходу инкрементальной компиляции, когда программа может обрабатываться поэтапно, по мере добавления в нее отдельных компонентов, а также модифицироваться в достаточно широких пределах «на лету», без повторной обработки фрагментов, не подвергшихся изменениям. Эти возможности образуют адекватный базис для построения на основе использования СП мощных языко-ориентированных процессоров различного назначения.

6.2. Валидация

Описанные выше методы `open()` и `create()` выполняют не только построение фрагментов СП, но и частичную проверку их корректности. Однако эта проверка осуществляется в минимально необходимом для продолжения генерации СП объеме. По существу, проверяется только структурная (синтаксическая) правильность входной конструкции (для функции `open()`) и корректность передаваемых функциям параметров. Функциональность полной семантической проверки компонентов СП вынесена в специальные виртуальные методы `validate()`, собственные версии которых определены для каждого класса СП, а их алгоритмы соответствуют семантике сущностей, представляемых классами-владельцами.

Каждая функция `validate()` проверяет семантическую корректность отношений между фрагментами (узлами) семантического представления, а также контролирует семантическую корректность представления в целом. Если в процессе работы функции обнаружены какие-либо нарушения корректности, соответствующий фрагмент СП помечается как ошибочный. Категория «ошибочность» понимается здесь в широком смысле и может трактоваться как «неполнота», «противоречивость» и т.п., в зависимости от семантики данного фрагмента СП. При последующих модификациях СП в случае успешного завершения повторного анализа фрагмента пометка будет автоматически удалена.

Несмотря на высокую сложность многих конструкций входного языка, предполагающих весьма нетривиальные алгоритмы семантических проверок, функции `validate()` работают достаточно эффективно: каждый вызов проверяет только те фрагменты СП, которые прямо или косвенно затронуты последними непроверенными модификациями. Повторного обхода заведомо правильных фрагментов не делается.

В заключение следует отметить, что в целом функциональность валидации аналогична традиционному этапу семантического анализа в

традиционных компиляторах. Существенное отличия заключаются, во-первых, в том, что в представленной архитектуре семантический анализ полностью отделен от стадии генерации СП, и, во-вторых, этот анализ реализуется по распределенной схеме. Последняя особенность будет коротко охарактеризована далее.

6.3. Валидация: распределенная схема

Как следует и предыдущих рассмотрений, в представляемой архитектуре валидация реализуется по «распределенной» схеме: семантические проверки реализуются в виде отдельных (виртуальных) методов, принадлежащих образам соответствующих конструкций. Такая организация существенно изменяет традиционный подход, когда семантический анализ, как правило, реализовывался в виде отдельного «прохода» компилятора по всему дереву программы.

Преимущества такого подхода заключаются в следующем. Во-первых, существенно повышается наглядность и сопровождаемость программного кода: каждый отдельный метод `validate()` отвечает за семантические проверки строго определенного класса конструкций языка. Все методы построены по единой последовательной схеме: сначала проверяется общая корректность конструкций, входящих в данную в качестве составных частей (путем вызова методов-валидаторов для их представлений), далее производится проверка собственной семантики данной конструкции. В большинстве случаев отдельные проверки имеют существенно унифицированный вид и представляют собой простые условные конструкции вида

если не *проверка свойства*
то *отметить конструкцию*
как ошибочную/неполную

Относительная простота и читаемость программного кода валидаторов позволяет рассматривать их как своего рода «эталонное» описание статической семантики конструкций входного языка, что сообщает этим методам самостоятельную ценность.

Во-вторых, распределенная схема дает возможность поэтапной реализации, когда семантические проверки могут свободно добавляться, модифицироваться и уточняться с течением времени. Семантическое представление в целом сохраняет работоспособность даже в случае наличия для некоторых (или даже всех) конструкций «пустых» валидаторов, что означает, естественно, полное отсутствие каких-либо проверок.

6.4. Семантический поиск

Эта особенность проектируемого семантического представления является одним из его наиболее важных свойств. Функциональность поиска, заложенная в СП, предоставляет исключительно мощные и в некоторых аспектах уникальные возможности для реализации весьма сложных манипуляций с программами.

Современные системы и среды программирования включают весьма ограниченные средства смысловой навигации по программам. Приведем фрагмент пожеланий создателя Си++ к развитой среде программирования, который касается функциональности запросов о программах:

«Моментальные ответы хотелось бы получать также на простые вопросы и указания типа: "Показать объявление `f`", "Какие еще `f` есть в области действия", "Как разрешен этот вызов оператора +?", "Какие классы произведены от `Shape`?" и "Какие деструкторы вызываются в конце этого блока?"»

Несмотря на очевидный прогресс в создании развитых средств поддержки программирования, сформулированные в этой цитате потребности в полной мере не реализованы ни в одной системе. По существу, единственным средством навигации, широко используемым в настоящее время, служит текстовый поиск по исходной программе с использованием механизма регулярных выражений. Такой поиск крайне ограничен по своим возможностям, не говоря уже о том, что он кодируется громоздкими и нечитабельными конструкциями.

Как легко понять, существо проблемы заключается в том, что такой поиск не является *семантическим*, так как использует только текстовое представление программы без анализа ее семантики. Так например, если запросы вида «найти все вхождения переменной `x` в заданном фрагменте текста» вполне по силам текстовому поиску, то для выполнения чуть более сложного запроса «найти все *использующие* вхождения переменной `x`, объявленной в классе `C`» потребуют составления весьма нетривиального регулярного выражения. А действительно сложные и в то же время практически полезные запросы вида «найти все классы из числа производных от класса `C`, в которых виртуальная функция `f` не перегружена» не могут быть представлены регулярными выражениями даже теоретически. В то же время с помощью

семантического поиска подобные запросы задаются естественно и элегантно.

Идея семантического поиска основывается на понятии шаблона и механизма сопоставления с образцом. Для каждой конструкции входного языка имеется «обобщенный» шаблон, отождествляемый с любой конструкцией данного вида, например, «условный оператор», «класс». При необходимости можно уточнить обобщенный шаблон, установив в нем нужные атрибуты, например, сформировав из обобщенного шаблона «класс» уточненный шаблон «класс с именем `C` из пространства имен `N`». Для задания более сложных образцов можно сконструировать суперпозицию обобщенных шаблонов, создав, например, «условный оператор с выражением «больше» в качестве условия». Наконец, можно воспользоваться стандартными функциями `create()`, чтобы сгенерировать произвольный шаблон. Вполне допустима ситуация, когда шаблон поиска представляет собой некоторое поддерево самой исходной программы, что позволяет искать в программе повторяющиеся фрагменты произвольной сложности. В общем случае шаблон поиска может представлять собой произвольную композицию обобщенных (предопределенных) шаблонов и конкретных фрагментов, сгенерированных стандартными средствами.

Собственно функциональность семантического поиска реализуется виртуальной функцией `match()`, версии которой определены для каждого узла семантического представления, то есть практически для любой конструкции входного языка. Чтобы найти, например, все вхождения операций присваивания в заданном составном операторе, достаточно вызвать для узла `COMPOUND_STATEMENT` его функцию `match()`, передав ей в качестве параметра шаблон «операция присваивания с произвольными левой и правой частями». Функция `match()` возвращает список элементов семантического представления, чья структура и атрибуты совпадают с заданным шаблоном.

Описанная модель дополнена возможностью задания дополнительных условий поиска программным путем (посредством спецификации значений тех или иных атрибутов или задания отношений между ними).

7. Ссылки

[1] Индекс популярности ЯП на ресурсе Tiobe Software: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Бьерн Страуструп, Дизайн и эволюция Си++: Пер. с англ.- М.: ДМК Пресс, 2000. ISBN 5-94074-005-7.

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

[4] ISO/IEC 15291:1999 Information technology -- Programming languages -- Ada Semantic Interface Specification (ASIS).

[5] Ада-компилятор GNAT на ресурсе http://www.adacore.com/home/products/gnatpro/.

[6] Веб-сайт компании Интерстрон: www.interstron.ru.

[7] Е.А. Зуев, Назначение и основные особенности компилятора Си++. Приложение к журналу КомпьюЛог, N3 (39), 2000, стр. 4-9.

[8] Веб-сайт проекта ROSE: http://www.rosecompiler.org/.

[9] The Pivot Reference Manual, http://parasol.tamu.edu/pivot/doc/ipr/index.html.

[10] Веб-сайт компании Edison Design Group: http://www.edg.com/.

[11] Инфраструктура CCI на ресурсе CodePlex: http://ccimetadata.codeplex.com/.

[12] E.Zouev, Common Compiler Infrastructure: A Compiler Writer's Perspective. Microsoft Research VSIP Workshop, June 26, 2003, Cambridge, England.