РЕАЛИЗАЦИЯ ПРОЦЕССОРА ОПТИМИЗАЦИИ КАК ИНСТРУМЕНТА СЕМАНТИЧЕСКИХ ПРЕОБРАЗОВАНИЙ ПРОГРАММ

Ходиев Ш. И.
к.т.н., доцент,
Национальный университет Узбекистана

Аннотация. Рассматриваются вопросы (автоматизации) создания процессоров оптимизации, и соответственно, исследования в области семантического анализа программ, позволяющие автоматизировать процессы создания инструментов преобразований, а также программные средства, обеспечивающие работу предложенных методов анализа. Инструменты могут значительно различаться, например, по характеристикам анализа, возможностями применения, обнаружению дефектов (полноте), точности и скорости поиска и другим. Рассмотрена сфера применения методов формальной семантики для разработки и реализации языков, создания на основе этих языков систем программирования. Значение работы определяется постоянно расширяющимися требованиями к статическим анализаторам, их возрастающей ролью в создании новых процессоров.

Ключевые слова. Процессор оптимизации, формальная семантика, инструментальные средства, статический анализ.

Введение. В процессе разработки и сопровождения программного обеспечения значительную роль играют инструментальные средства для проведения анализа аспектов времени выполнения приложений, их отладки, сопровождения. К первым относят корректное использование ресурсов, предоставляемых устройствами, эффективность реализации алгоритмов обработки данных и выполнения заявленной функциональности. Большинство из процессов обработки программ и данных реализуется в этих инструментах как текстовые или языковые, но является семантическими. В них, как правило, требуется сохранение некоторого инварианта, определенным образом связанного с семантикой обрабатываемых объектов (например, трансляция и другие функционально эквивалентные преобразования программ, которые сохраняют функцию, реализуемую программой [1]. Поэтому решение множества проблем информатики требует разработки формальных методов задания семантики языков.

Исследования, посвященные статическому семантическому анализу, являющегося основой большинства преобразований, ведутся во многих научных организациях и исследовательских центрах. Сложность определяется тем, что до сих пор не существует языка формального определения правил и модели правил, не выработано единой системы их классификации и отсутствует

быстрый универсальный анализатор. Остаётся актуальной задача разработки «алгебры программ», позволяющая манипулировать программными фрагментами в рамках формального исчисления программ с целью автоматизации конструирования эффективных и надежных программ для современных вычислительных устройств [2,3].

Статический анализ, заключающийся в исследовании кода без выполнения программы, является одним из подходов к созданию программ. Анализ статической семантики включает в себя потоковый анализ (анализ потоков управления и данных), анализ интерфейса, анализ перекрёстных ссылок и поиск потенциальных ошибок.

Известно, что нахождение точного решения задачи анализа потоков данных (т.е. наименьшего общего набора свойства при исполнении по всем путям до данной вершины) неразрешимо. Отсутствие точного метода описания семантики влечёт проблемы, в частности в понимании конструкций языков, в реализации этих языков. Задачами формальной семантики, помимо прочего, являются построение систем доказательств правильности программ, анализ свойств программ, например, проблем эквивалентности, завершимости программ, возможности их зацикливания. Это определило необходимость в создании семантических теорий, поддерживающих языки, или языков, обеспечивающих возможность применения имеющихся семантических теорий для анализа программ [1, 3-5]. Как развитие, отметим средства семантического поиска, под которыми понимаются системы, обрабатывающие запрос с использованием рассуждений над специфичной базой знаний, создание новых методов семантического анализа текстов, актуальных для решения многих задач в компьютерной лингвистике, таких как машинный перевод, классификация текстов и других [3-6]. Важным является разработка новых инструментов, позволяющих автоматизировать сам семантический анализ.

Математическую основу исследований составляют теория графов и алгебра логики. Существующие методы восстанавливают графы потока управления, графы вызовов функций, косвенные переходы и косвенные вызовы. Они включает в себя межпроцедурный, контекстно-чувствительный и потоко-чувствительный анализ исполняемых файлов.

Работа написана на основе проводимых на кафедре исследований, в том числе в рамках выпускных и магистерских диссертаций по созданию инструментов автоматизации программирования.

Статический анализ. Из-за расширения требований к статическим анализаторам для их удовлетворения привлекают новые подходы. Спектр используемых подходов весьма широк – от внутрипроцедурного анализа потока управления и данных до межпроцедурного анализа на основе аннотаций функций, чувствительного к путям выполнения анализа, символьного выполнения и определения выполнимости формул в теориях (SMT-решателям).

Могут осуществляться проверка статической семантики и стиля кодирования программ, а также некоторые другие сопутствующие анализу действия. В частности, статический анализ может выполнять функции документирования, форматирования, структурирования, инструментирования и измерения качественных характеристик программ.

В [4] рассмотрены методы анализа программ, применимые для проектов масштаба операционных систем и их наборов пользовательских приложений и реализованные в практически используемом анализаторе программного кода. В [3] рассмотрены методы анализа программ, применимые для проектов масштаба операционных систем и их наборов пользовательских приложений, и реализованные в практически используемом анализаторе программного кода. Кроме того, качественный анализатор должен масштабироваться для анализа больших файлов [6-8]. Статический анализ позволяет строить преобразователи, в том числе находить ошибки, которые трудно или невозможно обнаружить экспертным методом в силу размера современных программных комплексов и соответственно создавать мощные анализаторы различных типов, например анализаторы ошибок (исполнения). В последнем случае применяется инструмент статического анализа бинарного кода, позволяющий искать дефекты в уже готовой программе и в библиотеках. Соответственно, разработаны методы анализа значений и анализа помеченных данных, позволяющие проводить межпроцедурный, чувствительный к контексту и чувствительный к потоку данных анализ исполняемых файлов. Также, архитектуры системы анализа, методов и алгоритмов поиска клонов исполняемого кода, а также поиска дефектов. Результаты получены на базе использования методов абстрактной интерпретации и теории решеток.

В настоящее время известно множество методов поиска ошибок: экспертный аудит кода, ручное тестирование, дедуктивная верификация, динамический анализ (мониторинг выполнения программы, фаззинг и др.). Все эти методы имеют различные области применимости – не исключают, а дополняют друг друга. Одним из признанных методов поиска, широко используемым в индустрии, является статический анализ исходного кода программ. Анализатор, ищущий ошибки в программе, в результате работы предъявляет список мест в исходном коде, в которых найдены ошибки, с указанием типа ошибки, сообщения о ее характере и дополнительной информации о том, при каких условиях ошибка может произойти. С точки зрения анализатора границы между ошибкой, дефектом и уязвимостью достаточно условны. Все это – некоторые ошибочные ситуации в программе, то есть участки, дающие возможность предположить о нарушении семантики, неточности или неконсистентности модели программы, построенной анализатором. Классы ошибок, которые требуется искать, настолько разнообразны, что обойтись единственной моделью программы и возможно

точными алгоритмами анализа ее свойств на основе этой модели невозможно. Для таких ошибок, как, например, нарушения правил безопасного кодирования, неверное использование интерфейсов стандартных библиотек, требуется анализ абстрактного синтаксического дерева (АСД) и максимально детальная информация об исходном коде программы [7,9,10].

Используемое представление программного кода при статическом анализе программ играет важную роль. Различные варианты представления программ строятся на различных этапах компиляции. В [11] рассмотрено использование различных представлений java-программ для статического анализа.

Подходы к проведению анализа программ. Для решения этих проблем работают по следующим направлениям. Во-первых, разработали модели программы, пригодные для популярных императивных языков программирования (Си++, Java, C#), которые включают в себя модель памяти программы, единую для различных уровней анализа и настраиваемую для учета особенностей различных языков. Разработанные модели дали возможность вычислять необходимую информацию о значениях переменных программы с линейной масштабируемостью. Кроме того, математически обосновали алгоритмы анализа для построения моделей на следующих уровнях анализа: внутрипроцедурном анализе, межпроцедурном контекстно-чувствительном анализе всей программы, чувствительном к путям анализе. При этом вычисленная на предыдущих уровнях информация должна быть доступна для использования на последующих уровнях. На основе созданных моделей разработали алгоритмы поиска десятков классов ошибок (детекторы) – ошибок кодирования. Анализаторы, использующие глубокий межпроцедурный чувствительный к путям выполнения анализ, добиваются масштабируемости эвристическими алгоритмами, вносящими нестрогость в ход анализа [8]. В [9] предложена формальная модель ограничений, алгоритмов и структур данных программного средства для автоматической проверки исходного кода программ на соответствие ограничениям языков программирования С и С++ с малым временем работы, реализации разработанных средств в виде легковесного статического анализатора. Также разработка формальной модели ограничений и реализация на её основе модели памяти, удобной для проверки программ на соответствие различным классам ограничений, основанную на представлениях программы, используемых при компиляции. Разработаны алгоритмы статического анализа для быстрой проверки программ на соответствие рассматриваемому множеству ограничений, в том числе межпроцедурный метод обнаружения побочных эффектов, межмодульный метод анализа исключений.

Анализ объектно–ориентированных (ОО) языков. Примерами «неудобных» конструкций языков являются, например, такие средства

объектно-ориентированного программирования, как наследование, полиморфизм. Примерами «неудобных» требований окружения являются диагностика и компенсация ошибок. Реализация этих требований становится особенно сложной, именно при реализации оптимизирующих или распараллеливающих компиляторов, для расширяемых и специализированных языков, о многоязыковых системах. Такие сложности приводят, например, к тому, что при использовании средств автоматической генерации семантических анализаторов результатом генерации является не готовая программа, а некоторый полуфабрикат, который требует ручной доводки. В результате таких доводок нередко в реализацию вносятся ошибки. Большое число ошибок, как правило, встречается в блоках оптимизации и генерации кода. Кроме того, встречаются и системные ошибки, обусловленные сложностью интерфейсов между отдельными блоками компилятора [7].

Объектно-ориентированный анализ и проектирование приводят к объектно- ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, можно создавать гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний можно добиться большей уверенности в правильности разработанного программного обеспечения. В итоге уменьшается риск, характерный для разработки сложных программных систем.

В общем случае, конструкции ОО языков нельзя рассматривать как абсолютно новые. К ним применимы известные приёмы анализа и преобразований. Отсутствие требований структурированности (применительно к ОО отношениям) и прагматических критериев для таких программ, приводит к возрастанию роли языковых средств повышения эффективности. Из таких средств повышения качества ОО программ рекомендуются такие преобразования, как оптимизация кода программ, выбор нужной структуры данных, выбор нужного алгоритма, настройку исходного кода, использование быстрого языка программирования. Требуется введение дисциплинированного использования множественного наследования, что было предпринято при переходе от языка Си++ к JAVA. Когда иерархия классов становится достаточно многоуровневой, а замена методов в подклассах активно используемой, то программа становится плохо разбираемой. Появляются сложности, аналогичные тем, которые возникали при наличии переходов в неструктурированных программах. Предлагается использование (аналогичного способа) трансформирования свойства структурированности для ОО программ [5].

В работе [8] приводится описание алгоритма внутрипроцедурного анализа времени связывания для объектно-ориентированного языка. Приведены алгоритмы анализа времени связывания специализатора JaSpe для объектно-ориентированного языка Java. Анализ времени связывания в частичных вычислениях, нацеленных на оптимизацию программ, разделяет

программные конструкции на статические и динамические. Они применяются в основном для нетривиальной компиляции программ без компилятора, при наличии лишь интерпретатора и специализатора. Эффективность их существенно зависит от качества разметки программы. Одним из методов специализации являются частичные вычисления. Этот метод состоит из двух основных частей: анализа времени связывания и генерации остаточной программы. От аналогичных алгоритмов, работающих с императивными и функциональными языками, он отличается использованием объектов, которые позволяют получать более точную разметку — с большей долей статических конструкций — при обработке объектно-ориентированных программ.

Технологии реализации оптимизирующих преобразований (ОП). Существует достаточно много систем автоматизации, называемых системами построения трансляторов (СПТ) для экспериментов и создания промышленных (оптимизирующих) трансляторов. Один из способов включения ОП в процесс состоит в формулировании преобразований в терминах атрибутного дерева. Например, когда построение основывается на формальном описании трансляционной семантики реализуемого языка с помощью атрибутных грамматик. Тогда ОП становятся частью трансляционной семантики входного языка создаваемого транслятора. Метод разметки – основной метод анализа свойств состояний – не может впрямую применяться к атрибутным деревьям, поскольку в атрибутных грамматиках запрещены циклические зависимости. При втором способе разрабатывают одно или несколько промежуточных языков, добавляются этапы синтеза для перехода от атрибутного дерева к первому и следующим промежуточным языкам (ПЯ), упрощаются алгоритмы генерации. Возникает несколько этапов оптимизации по числу ПЯ [5,11].

Заключение. Существует много направлений исследований в области компиляторных технологий. Эти исследования используют интегрированную среду исследования алгоритмов анализа и трансформации программ. Оптимизации чаще разрабатывают как независимые от процесса трансляции, их можно менять, отключать не меняя алгоритмы трансляции. Используют ОП как трансформации, т.е. реализуют оптимизации методами ОП применением базовых трансформаций. Трансформационный подход (ТП) к оптимизации – это систематическое применение фундаментальных процессов семантической обработки программ. Модульный подход связан с разработкой одного или нескольких промежуточных языков для описания промежуточного (внутреннего) представления (ВП), на которой осуществляются ОП. Отсутствие алгоритма полной оптимизации не снижает практического значения задачи оптимизации и не ограничивает области теоретических исследований, так как реальные программы образуют довольно узкий класс, для которого можно получить теоретические и практические результаты.

ЛИТЕРАТУРА:

1. Касьянов В.Н. Оптимизирующие преобразование программ. – М.: Наука, 1988. – 336 с.
2. Бородин А.Е. Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++. Автореферат диссертации на соискание ученой степени кандидата физико-математических наук. Москва, 2016. –24 с.
3. Сидорин А.В. Метод межпроцедурного и межмодульного анализа кодов программ, написанных на языках С и С++, для построения многоцелевого контекстно-чувствительного анализатора. Автореферат диссертации на соискание учёной степени кандидата технических наук. МГТУ им. Н. Э. Баумана. Москва, 2017. – 20 с.
4. Белеванцев А.А. Многоуровневый статический анализ исходного кода для обеспечения качества программ. Автореферат диссертации на соискание ученой степени доктора физико-математических наук. Москва, 2017. – 42 с.
5. Ходиев Ш.И. Оптимизация циклов и удаление несущественных вычислений в оптимизирующих процессорах. Автореферат диссертации на соискание учёной степени кандидата технических наук. ТУИТ. Типогр. ТГТУ. Ташкент, 2010. –24с.
6. Архипова М.В.Генерация тестов для семантических анализаторов. Препринт ИСП. РАН, 2005. – 25 с.
7. Саргсян В.С.. Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода. Труды ИСП РАН, том 27, вып. 2, 2015. –С. 93-102
8. Igor A. Adamovich, Yuri A. Klimov. “The JaSpe specializer: an algorithm of intraprocedural binding time analysis for programs in Java language subset”. Program Systems: Theory and Applications, 2020, 11:1(44), pp. 3–29. (InRussian).
9. Орлов В.А., Клещев А.С. Компьютерные банки знаний. Многоцелевой банк знаний. Ж. Информационные технологии, 2006. №2.
10. Айк А.К. Методы статического анализа для поиска дефектов в исполняемом коде программ. Автореферат диссертации на соискание ученой степени кандидата физико-математических наук. Институт системного программирования им. В.П. Иванникова РАН. Москва, 2019. – 24 с.
11. Карпулевич Е.А. Использование различных представлений java-программ для статического анализа. Труды Института системного программирования РАН. 2015;27(6):151-158. https://doi.org/10.15514/ISPRAS-2015-27(6) –10.